ББК 32.973-018.1
УДК 681.3.06
К84

Крупник А. Б.

К84 Изучаем Ассемблер — СПб.: Питер, 2005. — 249 с.: ил.

ISBN 5-94723-757-1

Книга посвящена основам программирования на ассемблере в системах Windows и DOS. Знание ассемблера необходимо профессиональному программисту для понимания работы операционной системы и компилятора. Ассемблер позволяет написать программу (или ее часть) так, что она будет быстро выполняться и при этом занимать мало места. Это любимый язык хакеров; его знание позволяет менять по своему усмотрению программы, имея только исполнимый файл без исходных текстов. В основу изложения положены короткие примеры на ассемблере MASM фирмы Microsoft, вводящие читателя в круг основных идей языка, знание которых позволяет не только писать простые программы, но и самостоятельно двигаться дальше.

Книга рассчитана на школьников средних и старших классов, а также на всех интересующихся программированием вообще и ассемблером в частности.

ББК 32.973-018.1
УДК 681.3.06

ISBN 5-94723-757-1

Содержание

От автора

Ассемблер считается языком «крутых» программистов, и многие стараются выучить его, чтобы чувствовать свое превосходство над пользователями Паскаля или Бейсика. И в этом есть своя правда. Знание ассемблера позволяет понять внутреннее устройство программ и операционных систем, взаимодействовать с нестандартными устройствами, написать программу так, что она будет быстро работать и занимать мало места. Ассемблер — любимый язык хакеров; его знание позволяет менять по своему усмотрению программы, имея только исполняемый файл без исходных текстов. Знание ассемблера необходимо и для анализа вредоносных программ — компьютерных вирусов и червей, распространяемых через Интернет.

Казалось бы, все эти задачи, решаемые с помощью ассемблера, крайне сложны и требуют от программиста недюжинного ума. Но ассемблер, вопреки ожиданиям, совсем не сложен и выучить его гораздо проще, чем C++. Это обширный, со множеством инструкций, но прямолинейный и однозначный язык.

Разбирая примеры из книги, написанные для систем Windows и DOS, исследуя отладчиком получившиеся программы и пытаясь написать что-то свое, вы очень скоро поймете главное в ассемблере, его идею и суть, что позволит вам самостоятельно двигаться дальше.

Александр Крупник
krupnik@sandy.ru
11 сентября 2003 г.

От издательства

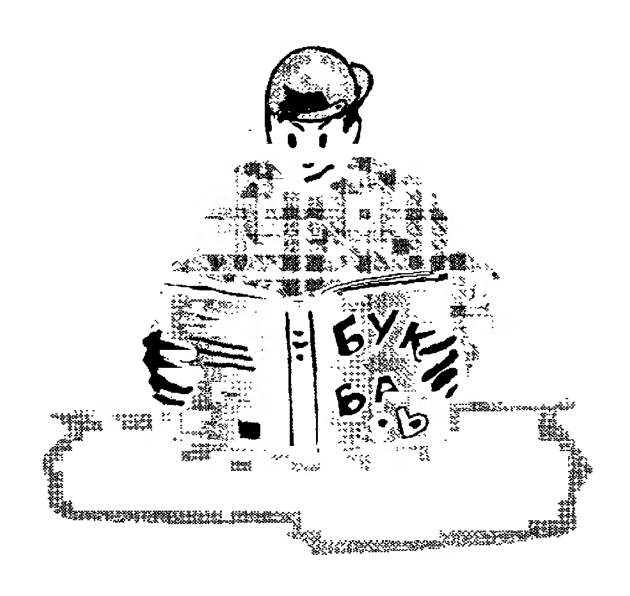
Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Все исходные тексты, приведенные в книге, вы сможете найти по адресу http://www.piter.com/download.

Подробную информацию о наших книгах вы найдете на web-сайте издательства http://www.piter.com.

ГЛАВА 1 Начало



Язык компьютера

> Я почти читаю ваши мысли: «Конечно, это компьютерная книга, и он пытается научить меня думать, как компьютер». Ничего подобного! Компьютеры думают, как мы. Ведь это мы их создали, как еще они могут думать? Нет, все, что я пытаюсь сделать, — это заставить вас пристально взглянуть на то, как вы думаете. Мы настолько привыкли все делать автоматически, что буквально не задумываемся над тем, как думаем.
>
> *Джеф Дантеман. «Ассемблер шаг за шагом»*

Ассемблер — родной язык компьютера. Можно сказать, что компьютер «думает» на ассемблере. Поэтому программы, написанные на других языках, таких как Си, нужно сначала перевести на ассемблер, чтобы компьютер их понял и смог исполнить.

Когда мы говорим о компьютере, выполняющем программы, то прежде всего имеем в виду его сердце — *процессор* — специальную микросхему, которая исполняет команды, часто называемые *инструкциями*, и хранит результаты их работы в специальных *регистрах*. Так, например, последовательность инструкций процессора

```
mov eax. 2
add eax. 3
```

приводит к тому, что в регистре eax оказывается число 5. Первая инструкция mov eax. 2 посылает в регистр eax число 2. Вторая инструкция add eax. 3, выполняемая вслед за первой, прибавляет к содержимому регистра eax число 3.

Операционная система

> Представьте себе абсолютно голого человека, выброшенного на необитаемый остров, и тогда вы поймете, что значит для нас жить без ДОСа под испепеляющим огнем процессора.
>
> *С. Расторгуев. «Программные методы защиты информации в компьютерах и сетях»*

Я немного лукавил, говоря в предыдущем разделе о том, что процессор понимает язык ассемблера без перевода. Процессор понимает только числа. Поэтому программа, написанная на ассемблере, должна быть переведена в числа, которые заставят процессор выполнить те или иные инструкции. Но каждой строчке ассемблера действительно соответствует одна инструкция процессора. Так что можно сказать, что ассемблер — это удобная человеку запись процессорных команд.

Чтобы перевести эту запись в понимаемые процессором числа, нужна особая программа, которая тоже называется ассемблером. Эта программа читает написанный человеком текст и затем переводит его в последовательность чисел, понимаемую процессором.

Текст программы на ассемблере содержит кроме инструкций еще и служебную информацию, которая помогает программе-ассемблеру понять, что же от нее требуется. Поэтому наша первая программа, прибавляющая к 2 число 3, будет выглядеть так, как в листинге 1.1.

Листинг 1.1. Первая программа

```
.386
.model  flat.stdcall
.code
start:
mov eax. 2
add eax. 3
ret
end start
```

В ней инструкции процессора mov, add, ret окружены директивами — специальными командами, которые должен выполнить не процессор, а сама программа-ассемблер. Первые три директивы нашей первой программы начинаются с точки.

Директива .386 показывает, для какого процессора предназначена программа. В нашем случае это процессор Intel 80386 и более поздние модели, ведь семейство процессоров Intel совместимо снизу вверх, и то, что умеет процессор 80386, под силу и процессорам 80486, Pentium, Pentium III, 4 и т. д.

Вторая директива .model flat, stdcall показывает, в какой среде будет «жить» программа. Дело в том, что программы работают не сами по себе, а под управлением *операционной системы*, которая их запускает и обеспечивает взаимодействие с внешней средой (вывод символов на экран, чтение и запись на жесткие диски и т. д.). В этой книге нас будет прежде всего интересовать операционная система семейства Windows 95[1], и директива .model... как раз и говорит о том, что именно для этой системы предназначена наша первая программа.

Третья директива .code показывает, где начинаются сами команды процессора. Когда операционная система пытается запустить программу, она ищет в ней инструкцию, с которой нужно начать, и отправляет ее под «испепеляющий огонь процессора». Когда же инструкции кончаются, операционная система «подхватывает» программу и помогает ей правильно завершиться, чтобы освободить место другим, ведь Windows — многозадачная операционная система, способная выполнять одновременно несколько программ. Уйти из-под опеки операционной системы помогает инструкция ret, стоящая последней в листинге 1.1.

[1] В это семейство входят системы Windows 95/98/ME/NT/2000/XP.

Инструкция, с которой начинается программа, обычно помечается последовательностью символов с двоеточием на конце (меткой). В нашем случае это start:. Там, где оканчивается последовательность команд процессора, в программе на ассемблере должна стоять директива end <метка первой инструкции программы>, в нашем случае это end start (после «start» двоеточие не ставится). Эта директива, а также сама метка никак не переводятся в инструкции ассемблера, а лишь помогают получить программу, которую способен выполнить процессор. Без них программа-ассемблер не поймет, с какой инструкции процессор начнет работу, и просто откажется работать.

Компилятор

Текст программы, показанный в листинге 1.1, предназначен для вполне определенной программы-ассемблера — это MASM фирмы Microsoft. Этот ассемблер чаще всего используется при разработке программ для Windows и к тому же он распространяется бесплатно.

Чтобы писать программы на ассемблере, одной программы-ассемблера мало, нужен еще редактор, в котором создаются и меняются тексты программ, а также удобная среда, в которой можно выполнять полученные программы, редактировать их и снова выполнять.

Такой средой для нас будет оболочка FAR, которую можно найти среди файлов к этой книге[1]. Установка оболочки очень проста: запускаем программу far165.exe

[1] Проще всего это делается так: ищете на сайте www.piter.com мою фамилию, в показанном списке книг выбираете книгу «Изучаем Ассемблер» и в ее кратком описании ищете раздел «файлы к книге». Там должны быть оболочка FAR, исходные тексты программ и компилятор.

и указываем папку, где она будет храниться. При первом запуске FAR (для этого нажимается кнопка Пуск и в программной группе Far Manager выбирается значок Far Manager) нужно указать удобный шрифт (я предпочитаю 10×18). Во всех шрифтах FAR есть латинские и русские буквы. Для перехода с латинского на русский используется правая пара клавиш Ctrl+Shift (держа Ctrl нажимаем Shift), для обратного перехода — левая пара клавиш Ctrl+Shift.

Теперь оболочку FAR можно использовать для установки компилятора. Пусть файл myasm.exe, найденный на сайте www.piter.com, находится в папке download на вашем диске C:. Предположим также, что файлы компилятора будут расположены в папке myasm на диске C. Если такой папки на диске C нет, ее нужно создать. Для этого нажимаем Alt+F1 и выбираем C в списке логических дисков. Далее нажимаем F7, в появившемся меню вводим название папки myasm и нажимаем Enter. Папка создана, и теперь нужно в нее перейти. Для этого передвигаем подсветку клавишами ↑↓, пока не выберется папка myasm. Еще одно нажатие Enter — и мы внутри. Теперь на панели справа нужно найти папку download диска C. Для этого нажимаем клавиши Alt+F2, выбираем диск C и далее — папку download. Теперь можно скопировать файл myasm.exe из папки c:\download в папку c:\myasm. Для этого подсвечиваем файл mayasm.exe, нажимаем F5 и затем — Enter.

Далее нужно перейти в папку f:\myasm и распаковать файлы компилятора. Для этого в оболочке FAR myasm.exe подсвечивается и нажимается Enter, после чего в папке myasm окажутся файлы компилятора.

Наша среда разработки программ на ассемблере почти готова. Осталось только указать путь к программе-ассемблеру, чтобы можно было запустить ее, находясь

в любой папке. Для этого в файле autoexec.bat[1] нужно изменить переменную path, добавив в нее следующую запись:

```
c:\myasm\bin
```

Вся строчка файла autoexec.bat, задающая переменную path, будет теперь выглядеть примерно так:

```
path=d:\util;f:\bcc55\bin;c:\myasm\bin
```

Как видите, все пути, кроме последнего, ведущего к файлам компилятора MASM, разделяются точкой с запятой.

Создание программы

Теперь мы наконец готовы создать первую работающую программу на ассемблере. Для этого нужен, прежде всего, файл, в котором будет храниться исходный текст программы, показанной в листинге 1.1. Чтобы создать такой файл, войдем в папку, где он будет храниться, нажмем клавиши Shift+F4, введем в появившемся окне имя файла (пусть это будет l11.asm)[2], введем текст программы из листинга 1.1, нажмем F10, выберем в появившемся меню пункт Save (сохранить) — и в нашей папке возникнет файл l11.asm. Посмотреть его содержимое можно клавишей F3. Для редактирования файла служит клавиша F4.

Нам остается создать еще один, так называемый *командный* файл, в котором содержатся команды программе-ассемблеру. Выглядит он так, как в листинге 1.2.

---

[1] В системе Windows XP может не быть файла autoexec.bat. Тогда путь к компилятору можно задать из панели управления: Панель управления ▸ Система ▸ Дополнительно ▸ Переменные среды.

[2] Каждый ввод последовательности символов или выбор пункта меню завершается клавишей Enter.

Листинг 1.2. Командный файл amake.bat

```
ml /c /coff "%1.asm"
link /SUBSYSTEM:CONSOLE "%1.obj"
```

К описанию команд этого файла мы вернемся чуть позже. А пока поместите его в одну из папок, указанных в переменной path. Самым удобным будет поместить файл amake.bat в папку, где хранятся небольшие утилиты, такие как архиваторы.

ВНИМАНИЕ

> Команда `amake 111` использует имя файла без расширения. Расширения `.asm` и `.obj` «приклеиваются» справа от имени 111, когда выполняется командный файл, и в результате получаются команды `ml /c /coff 111.asm` и `link /SUBSYSTEM:CONSOLE 111.obj`

После того как файл amake.bat создан и отправлен в подходящую папку, остается только перейти туда, где хранится исходный текст программы, набрать в командной строке оболочки (показана в нижней части рис. 1.1)

```
amake 111
```

и нажать Enter.

Если исходный текст программы набран без ошибок, то в папке, где он хранился, увидим два новых файла: l11.obj и L11.exe. Файл с расширением .exe и есть наша первая программа. А файл l11.obj — это «полуфабрикат», так называемый *объектный* файл, из которого получается готовая программа. Все дело в том, что текст больших программ хранится во многих файлах. Чтобы получить готовую программу, тексты на ассемблере сначала преобразуются в объектные файлы (в нашем командном файле это делает команда `ml /c /coff "%1.asm"`), а затем их обрабатывает *редактор связей* или *компоновщик*. В нашем случае редактор связей вызывается командой

```
link /SUBSYSTEM:CONSOLE "%1.obj"
```

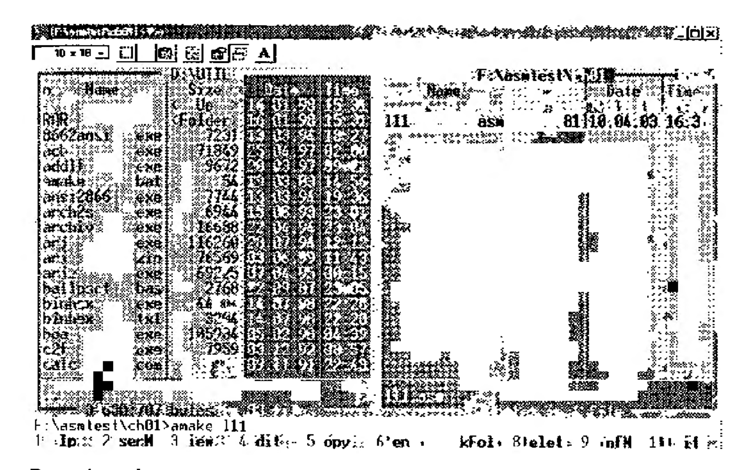



Рис. 1.1. До создания первой программы — одно нажатие Enter

И компоновщик, и программа, выдающая объектный файл (часто ее называют компилятором), управляются ключами — символами, стоящими непосредственно за косой чертой. Компилятор в нашем командном файле управляется двумя ключами: /c означает, что создается только объектный файл с расширением .obj, а ключ `/coff` определяет формат этого файла, стандартный для системы Windows. Компоновщиком управляет один ключ `/SUBSYSTEM:CONSOLE`, определяющий тип программы. В нашем случае это консольное приложение Windows, то есть программа, использующая для своей работы одно окно, куда она может выводить символы и откуда может эти символы читать. Консольными часто делают программы, управляемые ключами командной строки. Наш компилятор ml и компоновщик link — типич-

ные консольные приложения. Но вовсе не обязательно управлять консольными приложениями с помощью ключей командной строки. Ведь оболочка FAR с ее разветвленной системой меню (нажмите кнопку F9 — и увидите) тоже не что иное, как консольное приложение Windows.

Первые шаги

Описывая различные ключи компилятора и компоновщика, мы забыли о нашей программе l11.exe, оказавшейся в той же папке, где хранится текст на ассемблере l11.asm. Программа давно уже готова к тому, чтобы ее выполнил процессор. Для этого нужно подсветить имя l11.exe и нажать Enter. При этом голубые панели оболочки FAR на секунду исчезнут, приоткрыв черное пространство, куда выводятся результаты работы программы, и тут же сомкнутся над ним, не дав ничего толком разглядеть.

Впрочем, разглядывать особенно нечего, ведь наша программа только совершает действия с регистром процессора eax и в ней нет никаких команд вывода информации на экран. Приподняв клавишами CTRL+O голубые панели оболочки, увидим лишь командную строку l11.exe, говорящую о запуске программы, — и больше ничего.

Но это не значит, что результат ее работы скрыт от нас до тех пор, пока мы не научимся выводить символы на экран. Существует замечательная программа-отладчик OllyDbg, позволяющая увидеть программу изнутри и выполнить ее шаг за шагом.

Чтобы отладчик смог «подсмотреть» за программой, ее имя нужно передать ему в качестве параметра.

Набрав в командной строке FAR ollydbg l11.exe и нажав Enter, увидим множество окошек и ярлыков с непонятными значками (рис. 1.2), а также еще одно черное окно, куда программа должна выводить результаты своей работы. Но так как наша программа ничего не выводит на кран, окошко можно закрыть и сосредоточиться на отладчике.

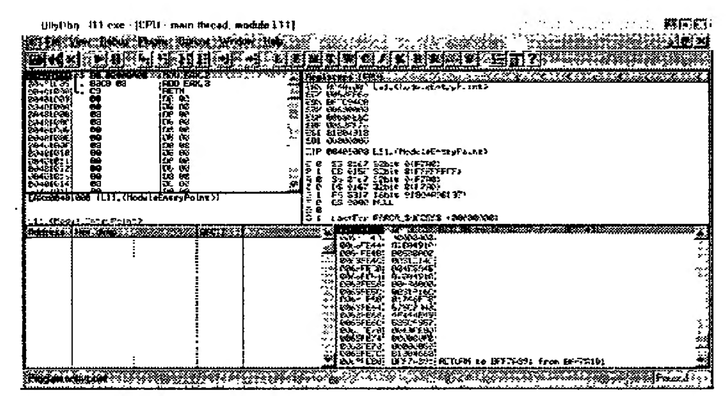


Рис. 1.2. Окно отладчика OllyDbg

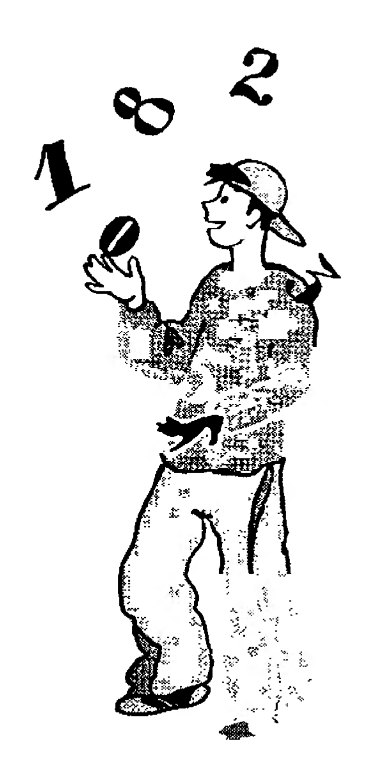
В нем инструкции программы расположены в левом верхнем углу (найдите там строчку mov eax, 2), а регистры процессора показаны в правом окне вверху (найдите значки EAX).

Начать пользоваться отладчиком, несмотря на его устрашающий вид, очень просто, потому что нам пока нужна только клавиша F8, выполняющая программу по шагам — инструкцию за инструкцией.

Нажав F8, увидим в правом окне, что регистр eax стал равен двум, а в левом окне подсвеченной оказалась уже вторая инструкция процессора add eax, 03. Нажав еще раз F8, увидим, что EAX стал равен 5. Это значит, что успешно выполнилась вторая команда add eax, 03.

И наконец, третье нажатие F8 приводит к тому, что выполняется команда ret, после которой программа должна покинуть операционную систему. Чтобы помочь ей, нужно выбрать мышью голубую стрелку ▸ в левом верхнем углу окна отладчика. Эта стрелка запускает программу не по шагам, а в обычном режиме. В нашем случае у программы уже нет собственных инструкций, и будут выполнены команды операционной системы, нужные для того, чтобы программа правильно завершилась.

ГЛАВА 2 Числа



8+8 = 10?

Наша первая программа, показанная в листинге 1.1, складывает 2 и 3, после чего в регистре eax оказывается число 5. Чтобы проверить этот результат, достаточно пальцев одной руки. Но давайте попробуем сложить два других числа — 8 и 8. Фундаментальные знания, полученные нами в первом классе, говорят, что здесь не хватит пальцев *обеих* рук. Но если скомпилировать программу, показанную в листинге 2.1,

Листинг 2.1. Сложение 8 и 8

```
.386
.model  flat,stdcall
.code
start:
mov eax, 8
add eax, 8 ;eax = 10????
ret
end start
```

и выполнить ее по шагам с помощью отладчика OllyDbg, то в регистре eax окажется число 10! Результат сложения двух восьмерок показан в листинге правее точки с запятой — символа, обозначающего начало комментария. Саму точку с запятой и все знаки справа от нее компилятор игнорирует. Комментарий помогает понять программу и предназначен не компилятору, а людям.

Но вернемся к результату сложения. Число 10 получилось не потому, что процессор ошибся, просто результаты его работы отладчик показывает в другой, *шестнадцатеричной* системе счисления, понять которую можно, задумавшись над устройством привычной нам десятичной системы.

Число 10_{10} (будем использовать нижний индекс для указания системы счисления, а если индекса нет, будем считать, что число записано в десятичной системе) устроено гораздо мудрее, чем это может показаться. 10_{10} — не просто последовательность двух символов — единицы и нуля, а краткая запись того, что число представляет собой сумму $\mathbf{1} * 10^1 + \mathbf{0} * 10^0$. Точно так же 4369_{10} — вовсе не картинка, не последовательность четырех символов, а краткая запись суммы $\mathbf{4} * 10^3 + \mathbf{3} * 10^2 + \mathbf{6} * 10^1 + \mathbf{9} * 10^0 = 4000 + 300 + 60 + 9$.

То есть любое число представляется суммой степеней десятки, что позволяет легко обращаться с такими числами: складывать, умножать, делить. Заметим, что коэффициенты при степенях десятки меняются от 0 до 9, что понятно: младший разряд числа, равный десяти, перестает быть младшим, это уже второй по значимости разряд и вместо $10*10^0$ следует писать $1*10^1$.

Говорят, что 10 — *основание* десятичной системы счисления, потому что все числа представляются в ней суммой степеней 10, а коэффициенты при степенях меняются от 0 до 9, то есть максимальный коэффициент на единицу меньше основания системы.

Естественно, ничто не мешает выбрать другое основание для системы счисления, например 16. Такая система во всем похожа на десятичную, только числа в ней представлены суммой степеней 16, а не десяти. Так, например, число 10_{16} — это сумма $\mathbf{1} * 16^1 + \mathbf{0} * 16^0 = 16_{10}$. То есть равенство 8 + 8 = 10 справедливо, если числа представлены в шестнадцатеричной системе.

Теперь следует подумать о том, как записывать шестнадцатеричные числа. Для записи числа 10_{16} хватило обычных арабских цифр. Но для коэффициентов при степенях 16 справедлива та же закономерность, что и в десятичной системе: минимальный коэффициент равен нулю, а максимальный — на единицу меньше основания системы счисления, то есть равен 15.

Как, например, представить число **2** * 16^1 + **15** * 16^0? Запись 215 не годится, потому что непонятно, где кончается один разряд (то есть коэффициент при степени шестнадцати) и где начинается другой. Ведь число 215, записанное таким образом, может быть равно **2** * 16^2 + + **1** * 16^1 + **5** * 16^0.

Внести определенность могли бы скобки, выделяющие каждый разряд: [2][15], но такая запись слишком неудобна при арифметических действиях из-за того, что разряды отличаются размером и числа трудно записать «столбиком».

Вот почему те разряды, для которых не хватает арабских цифр, принято обозначать буквами:

10_{10}=A: 11_{10}=B: 12_{10}=C: 13_{10}=D: 14_{10}=E: 15_{10}=F.

Это значит, что число **2** * 16^1 + **15** * 16^0 записывается в шестнадцатеричной системе как 2F, а число BAD равно **11** * 16^2 + **10** * 16^1 + **13** * 16^0 = 2989_{10}.

Двоящийся мир

Из-за ограниченности своих информационных резервуаров мне порой приходится выбрасывать отдельные символы, слова, предложения, накладывать тексты друг на друга так, что они претерпевают некоторые изменения, а для непосвященных теряют свою изначальную ясность. Но таковы законы Процессора, да не отсохнут у него разъемы.

С. Расторгуев. «Программные методы защиты информации в компьютерах и сетях»

Ясно, что в шестнадцатеричной системе можно записать любое число, но зачем? Ведь десятичная система удобней и привычней.

На этот вопрос можно ответить по-разному. Тот, кто скажет, что «таковы законы отладчика», будет, конечно, прав. Но отладчик поступает так не по прихоти, а потому, что шестнадцатеричные коды оказались самым лучшим посредником между человеком и компьютером.

Все дело в том, что процессор — это устройство, результаты работы которого — ряд значений напряжения на электрических контактах. Чтобы показать десятичное число, нужно десять градаций напряжения, а этого очень трудно добиться. Гораздо надежнее использовать всего две градации: ДА–НЕТ, Есть напряжение – Нет напряжения, 0–1. Но это означает, что числа, которыми процессору удобнее всего оперировать, должны быть представлены в *двоичной* системе!

Попробуем и мы, вслед за процессором, научиться оперировать двоичными числами. И прежде всего научимся переводить десятичные числа в двоичные. Сделать это довольно просто, если вспомнить, что в двоичной системе число должно быть представлено суммой степеней двойки. Так, например, 16_{10} = 2^4, следовательно, 16_{10} = **1** * 2^4 + **0** * 2^3 + **0** * 2^2 + **0** * 2^1 + **0** * 2^0 = 10000. Когда число не равно степени двойки, преобразование может быть более сложным, но все-таки легко понять, что 24_{10} = 16 + 8 = **1***2^4 + **1***2^3 + **0***2^2 + **0***2^1 + **0***2^0 = 11000, а 125_{10} = 64 + 32 + 16 + 8 + 4 + 1 = **1** * 2^6 +**1** * 2^5 + **1** * 2^4 + + **1** * 2^3 + **1** * 2^2 + **0***2^1 + **1** * 2^0 =1111101.

Двоичные числа так же хороши, как и десятичные. Скоро мы поймем, что двоичная арифметика гораздо проще десятичной. Есть, правда, одно неудобство при работе с двоичными числами: они слишком громоздки и трудно бывает понять, что за число скрывает длинный ряд нулей и единиц.

Но оказывается, что любое двоичное число можно компактно представить в шестнадцатеричном виде Чтобы понять это, посмотрим, какие числа можно хра-

нить в четырех двоичных разрядах. Минимальное число равно нулю, а максимальное — 1111, то есть $2^3 + 2^2 + 2^1 + 1 = 8 + 4 + 2 + 1 = 15$. Вспомнили? Ведь именно числа от 0 до 15 хранит любой разряд шестнадцатеричного числа! Значит, любые четыре идущих подряд двоичных разряда, или четыре *бита*, можно представить символами от 0 до F, используемыми при записи шестнадцатеричных чисел. То есть двоичное число 11111111 можно записать как FF, а число 11000001_2 — как C1.

Подчеркнем, что запись C1 — не просто сокращенное представление двоичного числа 11000001, а *настоящее* шестнадцатеричное число, равное $C1_{16} = 11000001_2 = 193_{10}$. Чтобы понять, почему так происходит, представим восьмиразрядное двоичное число в общем виде:

$p_7 * 2^7 + p_6 * 2^6 + p_5 * 2^5 + p_4 * 2^4 + p_3 * 2^3 + p_2 * 2^2 + p_1 * 2^1 + p_0 * 2^0$, где $p_7, p_6, p_5, p_4, p_3, p_2, p_1, p_0$ — двоичные разряды, равные нулю или единице. Поделим восемь бит на две равные части, называемые *тетрадами*. В младшую тетраду попадут биты 0–3[1], а в старшую — биты 4–7. Очевидно, старшую тетраду можно представить как

$p_7 * 2^7 + p_6 * 2^6 + p_5 * 2^5 + p_4 * 2^4 = 2^4 * (p_7 * 2^3 + p_6 * 2^2 + p_5 * 2^1 + p_4 * 2^0) = (p_7 * 2^3 + p_6 * 2^2 + p_5 * 2^1 + p_4 * 2^0) * 16^1$.

Число в скобках меняется от 0 до 15 и получается, что вторая тетрада двоичного числа принципиально не отличается от второго разряда числа шестнадцатеричного — разница только в обозначениях.

Подобные рассуждения можно повторить и для более длинных двоичных чисел, число разрядов которых

[1] Будем нумеровать биты согласно степеням двойки в представлении двоичного числа $\ldots p_3 * 2^3 + p_2 * 2^2 + p_1 * 2^1 + p_0 * 2^0$, то есть с нуля. Самый младший бит будет нулевым. Старший бит восьмиразрядного двоичного числа будет седьмым.

кратно четырем. Вывод очевиден: *чтобы перевести двоичное число в шестнадцатеричное, достаточно обозначить его тетрады символами 0–F, применяемыми для записи шестнадцатеричных чисел.*

Задача 2.1. Найдите способ автоматического перевода десятичных чисел в двоичные.

Подсказка: пусть вас вдохновит пример перевода десятичного числа в десятичное, показанный в табл. 2.1:

Таблица 2.1. Десятичные разряды числа (в обратном порядке)

Действие	Частное	Остаток
125 / 10	12	5
12 / 10	1	2
1 / 10	0	1

Конечность

В этом разделе мы узнаем самую, быть может, главную тайну компьютеров: все в них, оказывается, конечно. Конечна память на жестком диске, компакт-диске, DVD... Конечны и регистры процессора. О размере регистров можно догадаться, посмотрев их содержимое в окне отладчика.

Сумма 8 + 8, о которой говорилось в разделе «8 + 8 = = 10?» вовсе не равна 10_{16}, как мы до сих пор считали. Отладчик показывает число 00000010_{16}, и если вспомнить, что каждый шестнадцатеричный разряд соответствует четырем двоичным, то окажется, что в регистре eax умещается 32 бита.

Чтобы понять, много это или мало, найдем число состояний, в котором может находиться 32-разрядный регистр. Очевидно, нулевой (то есть самый младший) бит может быть в двух состояниях. Последовательность нулевого и первого бита имеет уже четыре состояния;

потому что на каждое из двух состояний нулевого бита приходится два состояния бита первого. Легко догадаться, что последовательность трех бит имеет 8 состояний, потому что на каждое из четырех состояний первых двух бит приходится два состояния третьего бита. Закономерность ясна: при добавлении бита число состояний удваивается, следовательно, 32 бита могут находиться в $2^{32} = 2^8 * 2^8 * 2^8 * 2^8 = 256 * 256 * 256 * 256 =$ 4294967296 состояниях. Четыре миллиарда двести девяносто четыре миллиона девятьсот шестьдесят семь тысяч двести девяносто шесть — весьма большое число, но это не избавляет нас от вопроса — что будет, если результат какой-то операции, например, сложения не уместится в 32 битах?

Очевидно, ничего хорошего. Но, программируя на ассемблере, нужно знать, когда возникает опасность и уметь отличить «правильную» операцию, результат которой верен, от «неправильной».

Для этого (и для многого другого) в процессоре фирмы Intel существует регистр флагов, некоторые биты которого показаны на рис. 2.1. Самый простой флаг — Z. Он поднимается (обращается в единицу), когда результат операции равен нулю.

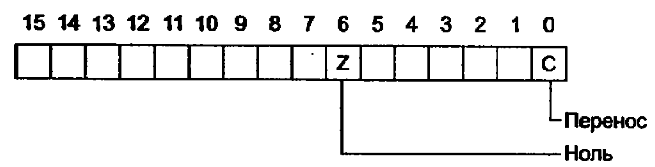


Рис. 2.1. Флаги переноса и нуля

Чтобы понять роль второго флага, попробуем сложить два одинаковых больших числа, равных в десятичной системе 4000000000. Программа, которая это проделывает, показана в листинге 2.2.

Листинг 2.2. Сложение двух больших чисел

```
.386
.model flat,stdcall
.code
start:
mov eax, 4000000000
add eax, 4000000000
ret
end start
```

Сумма двух таких чисел равна 8000000000, но мы уже знаем, что в 32-битовом регистре может поместиться число чуть большее 4000000000. Поэтому будет любопытно скомпилировать программу командой amake l22, запустить отладчик и посмотреть результат. На рис. 2.2 показано состояние программы после выполнения двух первых команд — mov и add.

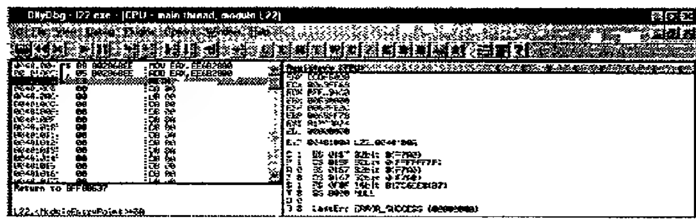


Рис. 2.2. Результат сложения двух слишком больших чисел

Видно, что оба слагаемых представлены в шестнадцатеричной системе как EE6B2800, а их сумма равна DCD65000. Кроме того, поднялся флаг переноса, обозначенный буквой C в нижней части правого окна отладчика, а флаг Z, наоборот, опустился (обратился в ноль), потому что результат операции сложения — явно ненулевой.

Оба наших слагаемых меньше предельного числа, способного уместиться в регистре. Значит, их шестнадцатеричное представление, показанное отладчиком,

верно. А вот сумма не может поместиться в 32 бита. И чтобы посмотреть, где «ошибся» процессор, попробуем сложить два числа вручную (рис. 2.3).

```
 EE6B2800
+EE6B2800
1DCD65000
```

Рис. 2.3. Сложение шестнадцатеричных чисел

Складывать шестнадцатеричные числа труднее, чем десятичные. Но лишь потому, что мы не знаем таблицы шестнадцатеричного сложения. Попробуем в качестве тренировки сложить два числа, показанные на рисунке.

Как обычно, начинаем с младших разрядов, и первые два сложения очевидны, ведь 0 + 0 = 0 в любой системе счисления. Далее идет сложение 8 + 8, что дает в десятичной системе 16. Но 16 — основание шестнадцатеричной системы, поэтому 8 + 8 — это «0 пишем, один в уме». Этот «один в уме» называется переносом в старший разряд, что сделает сумму следующих 2 + 2 равной 5 (2 + 2 + перенос). Теперь нам необходимо сложить B + B. Поскольку таблицы сложения мы не знаем, приходится соображать, что $B + B = 22_{10} = 16_{10} + 6$, то есть «шесть пишем, один в уме». Продолжая в том же духе, получим сумму, показанную на рисунке. Она отличается от той, что показал отладчик, единичкой в самом старшем, 33-м, разряде. Складывая столбиком, мы не теряем разрядов, если, конечно, слева остается бумага. Но в регистрах всего 32 бита, поэтому процессор, заметив, что есть перенос из старшего разряда, устанавливает в единицу флаг С. Можно сказать, что бит, не поместившийся в регистре, сваливается с левого конца регистра и сохраняется во флаге переноса. Вот почему в нашем примере флаг С равен единице!

Знак

До сих пор мы думали, что складываем положительные числа — просто потому, что не знали никаких других. На самом деле, любое число для процессора — всего лишь последовательность двоичных разрядов — бит. Поэтому в регистре, состоящем из 32 бит, можно закодировать 2^{32} разных чисел, а какими они будут — положительными, отрицательными, целыми или дробями — зависит от договоренности.

Попробуем понять, как в компьютере кодируются отрицательные числа, для чего перейдем от настоящих 32-битовых чисел к «игрушечным» 4-битовым. Такими числами гораздо проще оперировать, а то, что удалось понять на их примере, легко обобщить на любое число двоичных разрядов.

Итак, регистр, состоящий из 4 бит, может находиться в $2^4 = 16$ различных состояниях, и логично поделить его пополам: 8 состояний (включая 0) будут «положительными», 8 — «отрицательными». Чтобы отрицательное число сразу можно было отличить от положительного, сделаем старший, четвертый, бит знаковым; пусть он будет равен нулю для положительных чисел и единице — для отрицательных.

С учетом сказанного в четырех битах могут уместиться (вместе с нулем) такие положительные числа:

0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Отрицательные числа легко получить из положительных, если учесть, что сумма положительного числа и такого же по абсолютной величине отрицательного равна нулю. Если просто обратить все биты положительного числа, записав вместо 0 — 1, а вместо 1 — 0, то все биты суммы окажутся равными единице, например,

0001 + 1110 = 1111.

Здесь используется нехитрое правило сложения двоичных разрядов 1 + 0 = 0 + 1 = 1.

А теперь представим себе, что к сумме, состоящей из единиц, добавляется еще одна единица. Согласно правилам двоичной арифметики, 1 + 1 = «ноль пишем, один в уме». Ведь 1 + 1 — это два — основание двоичной системы счисления, поэтому в младшем разряде пишется 0, а единица переносится в старший разряд (в десятичной системе этому соответствует сумма 5 + 5, которая тоже равна «ноль пишем, один в уме»).

Это значит, что единичные биты от прибавления еще одной единицы «повалятся», превратятся в нули, и результатом сложения будет единица, но уже в пятом, несуществующем для 4-битовых чисел разряде:

–1 + 1 = 1111 + 0001 = 10000

То есть в наших четырех битах окажутся нули, что и требуется. Итак, согласно правилу обращения битов и добавления единицы, отрицательные числа будут кодироваться так:

–1 1111
–2 1110
–3 1101
–4 1100
–5 1011
–6 1010
–7 1001

Теперь у нас есть коды 15 чисел (0, 7 положительных и 7 отрицательных). Всего в 4 битах умещается 16 чисел, поэтому прибавим к ним код для –8. Его нельзя получить обращением битов, потому что нет соответствующего положительного числа. Но можно воспользоваться тем, что сумма положительного числа и соответствующего отрицательного для регистра из четырех битов всегда равна 16. Чтобы, например, получить двоичный код для –7, необходимо представить в виде двоичного числа разность $16 - 7 = 9_{10} = 1001_2$. Проверка показывает, что $7 + (-7) = 0111 + 1001 = 10000 = 16_{10}$, что и требуется. Поступая с восьмеркой так же, как только что с семеркой, получим:

–8 = 16 – 8 = 8 = 1000

Только что полученный код для отрицательных чисел называется *дополнительным*, потому что отрицательное число получается дополнением положительного до 16[1].

Этот код обладает многими замечательными свойствами. Во-первых, в нем существует только один ноль, ведь –0 = 1111 + 1 = 10000 = 0, потому что пятый единичный бит не умещается в 4-битовом регистре и пропадает.

Во-вторых, знак числа можно менять бесконечное число раз без каких-либо изменений и потерь. Чтобы поменять знак числа –5, нужно обратить[2] все биты числа 1011 и прибавить единицу: $0100 + 1 = 0101 = 5_{10}$. Затем можно получить –5 — и так до бесконечности.

И, наконец, в третьих, дополнительный код позволяет свести вычитание к сложению. Чтобы, например,

---

[1] Дополнение до шестнадцати справедливо только для 4-битовых регистров. Для 8- 16- и 32-битовых регистров числа будут иными, но принцип сохранится.

[2] Обращение битов, то есть запись 1 вместо 0 и нуля вместо 1 часто называют инвертированием.

вычесть из пяти три, достаточно записать в регистр 3, затем инвертировать все биты, прибавить единицу, после чего в регистре окажется число –3 в дополнительном коде, и затем уже прибавить пять. Это удобно процессору, потому что операции инвертирования и сложения для него очень просты.

Переполнение

Числа, с которыми мы познакомились в предыдущем разделе, стремятся вырваться за пределы четырех битов, отчего многие результаты действий с ними оказываются неверными.

Ясно, что в 32, 16 и даже 8 битах места гораздо больше, но и там нужно уметь определять, когда результат операции верен, а когда нет. Попробуем узнать границы дозволенного для 4 бит, с надеждой применить полученные знания к «реальным» числам, обитающим не в тесных 4-битовых клетках, а в просторных, но все же конечных 32-битовых вольерах.

Прежде всего заметим, что сложение чисел с разным знаком всегда безопасно, потому что абсолютная величина числа при этом уменьшается и выхода за пределы отведенных ему бит не происходит.

Когда складываются числа с одинаковым знаком, все не так. Сумма 7 + 5 дает 12 — число, которое не помещается в 4 битах. Переходя к двоичным кодам, видим, что знак суммы в этом случае не такой, как у слагаемых (рис. 2.4).

```
  7+5          -7 + -5
  0111           1001
+ 0101         + 1011
  ----          -----
  1100          10100
```

Рис. 2.4. При «неправильном» сложении сумма и слагаемые отличаются знаком

Получается, что 7 + 5 = –4. Точно так же меняется знак суммы при «запрещенном» сложении двух отрицательных чисел –7 + –5. То есть сумма –7 и –5 получилась равной 4, если не учитывать пятый единичный бит, вытесненный за пределы 4-битового регистра.

Отличие знака суммы от знака слагаемых называется *переполнением*. На такое событие реагирует флаг O (от слова overflow, переполнение), показанный на рис. 2.5. При переполнении флаг O обращается в 1. Изменение знака суммы покажет и флаг знака S (рис. 2.5), который поднимается, когда результат операции отрицателен, и опускается, когда тот положителен.

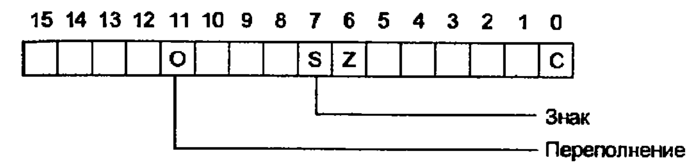


Рис. 2.5. Флаги переполнения и знака

Заметим, что условие переполнения (отличие знака суммы от знака слагаемых) не связано с размером регистров и потому применимо к любому числу битов: 8, 16, 32.

Задача 2.2. Докажите, рассмотрев все возможные варианты сложения в пределах 4 бит, что переполнение возникает, лишь когда есть перенос из старшего бита, но нет переноса в старший бит, либо наоборот — когда есть перенос в старший бит, но нет переноса из старшего бита.

Нам осталось понять самое главное: процессор ничего не знает ни о положительных, ни об отрицательных числах. Он способен только тупо складывать биты по правилу 0 + 0 = 0, 0 + 1 = 1 + 0 = 1, 1 + 1= «ноль пишем, один в уме». Что такое 1111_2 — положительное число 15

или же –1 в дополнительном коде, — знает только программист.

Числа EE6B2800, сложением которых мы занимались в разделе «Конечность», можно рассматривать как большие положительные числа 4000000000_{10}, и тогда результат сложения неверен, потому что не может уместиться в 32 битах. Об этом говорит флаг переноса C, равный 1.

Но теми же битами записывается в дополнительном коде число –294967296, и тогда результат сложения DCD65000 верен, равен -589934592_{10} и легко умещается в 32 битах, о чем и говорит флаг O, установленный в ноль.

Теперь понятно, что флаг C сигнализирует о неправильном сложении беззнаковых, положительных чисел. А флаг O показывает, что неверен результат сложения чисел со знаком.

Байты и слова

К сожалению, 4-битовых регистров, столь удобных для изучения двоичной арифметики, не бывает. Минимальное число доступных процессору бит равно восьми. Эти 8 бит называют *байтом* и делят на две равные части — тетрады, в каждой из которых 4 бита. Состояние каждой такой четверки удобно задавать шестнадцатеричным кодом. Например, байт, в котором все двоичные разряды равны единице, задается символами FF.

До сих пор мы считали регистры процессора монолитными. На самом же деле четыре регистра: eax (уже известный нам), а также ebx, ecx и edx можно поделить пополам, а одну из половин — еще раз пополам. В результате получится, что в каждом из этих четырех регистров окажется доступным малый 16-битовый регистр, а в нем — два байта (рис. 2.6).

31	23	15	7 0	16-BIT	32-BIT
		AH	AL	AX	EAX
		DH	DL	DX	EDX
		CH	CL	CX	ECX
		BH	BL	BX	EBX

Рис. 2.6. В регистрах поселились байты и слова

Так, например, в регистре edx доступно 16-битовое слово dx, а в нем — два байта — dh (старший) и dl (младший). Как показывает листинг 2.3, с этим словом и байтами можно обращаться так же, как и с целым регистром.

Листинг 2.3. Пример работы с 8- и 16-битовыми регистрами

```
.386
.model flat,stdcall
.code
start:
mov al, -120        ;al = 88h
mov bl, -127        ;bl = 81h
add al, bl          ;al = 09h O = 1 C = 1 S = 0
                    ;то же сложение,
                    ;но в регистрах AX,BX
mov ax, -120        ;al = 88₁₆
mov bh, 255         ;bx = ff81₁₆ = -127
add ax, bx          ;ax = ff09₁₆ = -247 O = 0 S = 1
ret
end start
```

Программа из листинга 2.3 начинается с попытки сложить два числа –120 и –127, хранимых в байтах al и bl. Чтобы понять, имеет ли смысл такая операция, выясним, какие числа умещаются в восьми битах. Очевидно, байт или 8 бит способен быть в $2^8 = 256$ различных состояниях. По аналогии с 4-битовым регистром можно понять, что байт хранит числа от –128 до 127.

Значит, сумма –120 и –127 никак в нем не уместится. Поэтому операция add al, bl вызовет переполнение и флаг O станет равен 1[1].

Чтобы сложить числа –120 и –127, воспользуемся 16-битовыми регистрами ax и bx. Содержимое al у нас уже испорчено операцией сложения, поэтому поместим в регистр ax число –120 командой mov ax, -120.

Содержимое bl не пострадало. Поэтому попробуем сообразить, чему должен быть равен байт bh, чтобы содержимое регистра bx стало равно –127.

Тут нужно вспомнить, что дополнительный код, в котором записываются отрицательные числа, зависит от числа битов в регистре. Для 4-битового регистра отрицательное число получается дополнением до 16, то есть до 2^4 — числа различных состояний, в которых может находиться последовательность из 4 бит. Для 8-битового регистра необходимо уже дополнение до $2^8 = 256$. А для 16-битового регистра это уже дополнение до $2^{16} = 2^8 * 2^8 = 256 * 256 = 65536$. Но вспомним, что в регистре bl уже есть дополнение до 256, ведь там хранится отрицательное число –127. Значит, нам осталось дополнить 16-битовый регистр до 65536 – 256 = 65280, что равно в шестнадцатеричном представлении FF00. То есть все разряды старшего байта должны быть равны единице! Вот зачем нужна инструкция mov bh, 255 ($255 = ff_{16}$) в листинге 2.3.

Итак, для переноса отрицательного числа в более просторный регистр нужно все биты, стоящие левее знакового, сделать равными единице. Такая операция

[1] Важно понимать, что байты al, ah и т. д. выступают в арифметических операциях как самостоятельные регистры. Казалось бы, al и ah — только части регистра ax, а тот, в свою очередь, — лишь часть eax. Но перенос из старшего разряда al не попадает в ah! Он только поднимает (устанавливает в 1) флаг C.

называется *расширением знака*. Очевидно, для переноса положительного числа нужно все старшие биты приравнять нулю. Пусть, например, в байте al хранится число 100. Приравняв к нулю байт ah, мы добьемся того, что число 100 займет 16-битовый регистр ax. Если же в al хранится –100 в дополнительном коде, то приравняем ah = ff и число –100 переселится в более просторный регистр ax.

Завершим этот раздел двумя важными замечаниями. Первое касается перевода из одной системы счисления в другую. Для этого проще всего использовать программу «Калькулятор» системы Windows. Выбрав в меню Вид пункт Инженерный, увидим на экране примерно то же, что и на рис. 2.7.

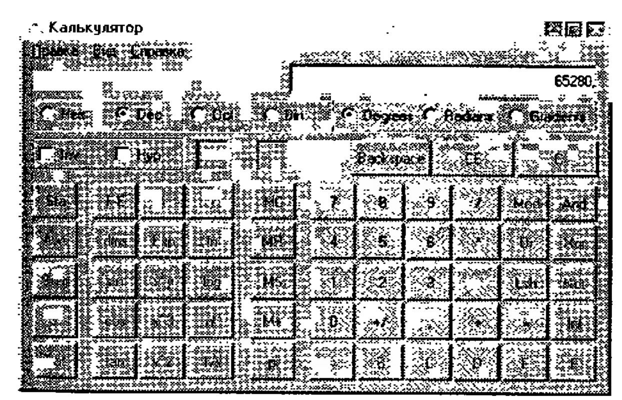


Рис. 2.7. Перевод числа из десятичной (dec) системы

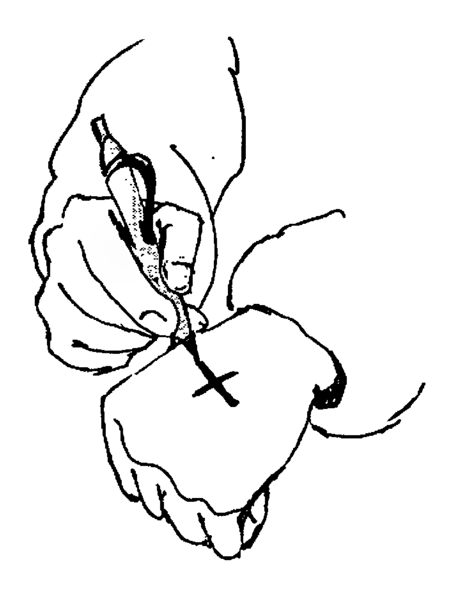
Далее следует указать систему счисления (на рисунке выбрана десятичная система), ввести число и затем выбрать мышью другую систему, в которую хочется перевести заданное число. Калькулятор знает о двоичной (Bin), шестнадцатеричной (Hex) и восьмеричной (Oct) системах счисления.

Второе замечание касается записи чисел в программах на ассемблере. Оказывается, можно использовать не только десятичные, как мы это делали до сих пор, но и шестнадцатеричные и двоичные числа. Например, инструкция mov bh,255 может быть записана как mov bh, 0ffh, где буквой h помечается шестнадцатеричное число, или как mov bh, 11111111b, где буква b обозначает двоичное число. Все числа независимо от системы счисления должны в программах на ассемблере начинаться с цифры, поэтому перед ff стоит ноль. И наконец, десятичные числа ничем не лучше других, поэтому их следует помечать буквой d: mov ah, 255d. До сих пор мы этого не делали, потому что ассемблер по умолчанию считает числа десятичными. Но можно изменить предпочтения ассемблера директивой .radix. Если в начале программы стоит

```
.radix 16
.386
.model  flat,stdcall
start:
mov al, -120d
…
```

ассемблер будет считать все числа без «опознавательных знаков» шестнадцатеричными и тогда буква d станет обязательной для каждого десятичного числа.

ГЛАВА 3 Память



Адреса

Программы, выполняемые процессором, находятся не в воздухе и даже не в самом процессоре, а в оперативной памяти компьютера. Процессор забирает из памяти очередную команду, выполняет ее, потом переходит к следующей команде, снова выполняет ее — и так до конца программы. Команды процессора могут не только менять содержимое его регистров, но и записывать числа в память компьютера, состоящую из отдельных, идущих друг за другом байтов.

Все байты компьютерной памяти пронумерованы. Самому первому присвоен нулевой номер. Номер последнего байта определяется объемом оперативной памяти, которой располагает компьютер. Номер байта обычно называют *адресом*. Адреса команд и данных, хранящихся в памяти, всегда видны в окне отладчика, нужно только научиться их замечать. Поможет в этом программа, показанная в листинге 3.1.

Листинг 3.1. Взаимодействие с памятью компьютера

```
.386
.model  flat.stdcall
.data
data_8      db -3
data_16     dw ?
.code
start:
mov al. data_8
sub ah. ah
dec ah
mov data_16. ax
ret
end start
```

В программе использована директива .data, указывающая процессору, что следом за ней идут *данные* — числа, символы, словом, все то, что нельзя считать командами процессора. Ассемблер будет считать данными все, что расположено в исходном тексте программы до директивы .code.

Как и регистры процессора, данные отличаются размером. Директива db задает байт памяти, директива dw — слово (два идущих подряд байта), директива dd — двойное слово или четыре байта.

Запись data_8 db -3d означает, что в области памяти под именем data_8 хранится байт –3. Запись sum dw ? выделяет память для двух идущих подряд байтов (слова), знак вопроса показывает, что значение байтов заранее не определено. При запуске программы там может быть все что угодно.

Инструкция mov al. data_8 берет из памяти байт, помеченный как data_8, и записывает его содержимое в регистр al. При этом содержимое байта data_8 не страдает, он как бы размножается, ведь после выполнения инструкции число –3 оказывается не только в памяти, но и в регистре al.

Инструкция sub ah. ah посылает разность ah - ah в регистр ah. Каким бы ни было содержимое ah, там после такой операции окажется 0. Наконец, инструкция dec ah уменьшает содержимое ah на единицу. А поскольку там до этого оказался 0, то в результате получится 0 – 1 = –1 или 0FFh в шестнадцатеричном представлении. Вместо инструкции dec можно было бы написать sub ah. 1, но такая запись длиннее и программисты гораздо реже ее используют.

Инструкция dec, обращающая все биты ah в единицу, расширяет знак числа –3, попавшего в al (см. раздел «Байты и слова» главы 2). После нее число –3 переселяется в регистр ax, откуда пересылается инструкцией

mov data_16, ax в область памяти data_16, состоящую из двух идущих подряд байтов.

Чтобы «почувствовать» адреса памяти, полезно увидеть результаты работы программы в окне отладчика (рис. 3.1.)

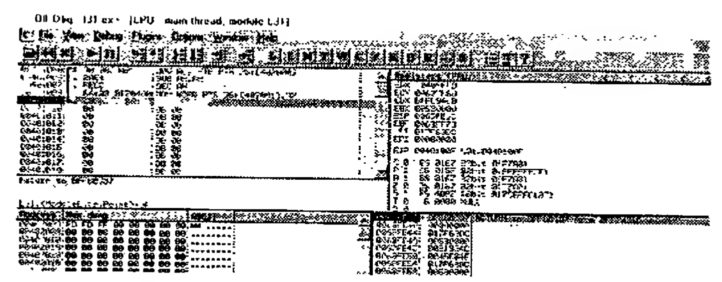


Рис. 3.1. Итог работы программы

В левом верхнем углу видны адреса памяти, занимаемой командами процессора. Первая инструкция располагается в памяти, начиная с адреса 00401000[1], и занимает 5 байт. Вторая инструкция sub ah, ah умещается в двух байтах с адресами 00401005, 00401006. Наконец, последняя инструкция ret занимает всего один байт и находится по адресу 0040100f.

Область данных хоть и помещена в листинге 3.1 раньше команд, имеет большие адреса, видные в левом нижнем окне отладчика. Байт data_8 имеет адрес 0040200 и хранит число –3 или fd в шестнадцатеричном виде. Следующие два байта помечены в листинге словом data_16 и должны хранить FFFD — число –3, записанное в дополнительном коде. Но отладчик показывает, что первым идет байт FD (его адрес 0040201)), а следом уже байт FF (его адрес 0040202). Так происходит потому, что в процессорах Intel числа располагаются в памяти по правилу: *младший* байт имеет *меньший* адрес.

[1] Все адреса записаны в шестнадцатеричном коде.

Это важнейшее правило позволяет нам узнать много нового об устройстве команд процессора. Взгляните еще раз на первую команду mov al, data_8, занимающую пять соседних байтов в памяти компьютера:

```
mov al, data_8    A0 00204000
```

Очевидно, четыре идущих подряд байта 00 20 40 00 — это вывернутый наизнанку адрес числа data_8. Читая их справа-налево, то есть в обратном порядке, получим 00402000 — адрес, видный в левом нижнем окне отладчика. Байт с таким адресом равен FD, то есть –3. В листинге 3.1 он помечен как data_8.

Очевидно, любая команда процессора уже в самом первом своем байте содержит информацию о ее типе и длине (в нашем случае это байт A0). Только так можно устранить путаницу и всегда отличать конец одной команды от начала другой.

В этом разделе мы пересылали числа только из памяти в регистр. Наверняка многие сразу захотят использовать команду

```
mov data1, data2; Так не бывает!!!
```

Но ассемблер откажется переводить это вздорное требование на язык процессора, потому что *хотя бы один операнд инструкции* mov *должен быть регистром.* Такое ограничение связано с устройством процессора и его взаимодействием с памятью компьютера. Знание деталей этого устройства не поможет программисту преодолеть ограничения команды mov. Все равно для пересылки из памяти в память придется использовать промежуточный регистр или какие-то другие инструкции процессора или же специальную область памяти, называемую *стеком.*

Стек

При сохранении данных в памяти компьютера не обязательно указывать адрес. Специальная команда процессора push помещает слово или двойное слово в область памяти, называемую стеком, а команда pop читает данные из стека и записывает их в регистр или «обычную» область памяти. С помощью команд push можно «натолкать»[1] в стек множество чисел, но команда pop вернет оттуда число, *помещенное в стек последним*. Следующая команда pop вернет из стека число, которое втолкнули предпоследним, и если заставить процессор выполнить столько же команд pop, сколько было команд push, то все числа вернутся назад, но в обратном порядке: число, помещенное в стек первым, вернется последним. Иллюстрирует сказанное программа, меняющая содержимое регистров eax и ecx (листинг 3.2).

Листинг 3.2. Регистры меняются содержимым через стек

```
.386
.model  flat.stdcall
.code
start:
mov eax. 2 ;eax = 2
mov ecx. 3 ;ecx = 3
push eax
push ecx
pop  eax   ;eax = 3
pop  ecx   ;ecx = 2
ret
end start
```

Первая команда push eax посылает в стек содержимое регистра eax. Следующая команда push ecx сохраняет в стеке регистр ecx, то есть число 3. Команда pop eax выталкивает из стека число, помещенное туда последней командой push. В нашем случае это число 3.

[1] Push в переводе с английского и значит «толкать».

Значит, после команды pop eax в регистре eax появится число 3 — точно такое же, как в регистре ecx. Но это равенство сохранится недолго. Следующая команда pop ecx опустошает стек, выталкивая из него число 2 и помещая его в регистр ecx. Выходит, что перед завершением программы ecx стал равен двум, а eax — трем. То есть регистры благодаря стеку обменялись содержимым.

Замечательно то, что при таком обмене ничего не нужно знать о внутреннем устройстве стека. Достаточно двух простых правил:

1. Помещенное в стек последним выходит первым.
2. Размер переменных, помещаемых в стек и доставаемых оттуда, должен совпадать.

Если бы нам вздумалось поменять значения регистров ax и ecx командами

```
push ax
push ecx
pop  ax
pop  ecx
```

то ничего хорошего из этого бы не вышло, потому что стек не может уместить содержимое 4-байтового регистра в 2-байтовом. Команда pop ax заберет из стека последние два байта и разорвет регистр ecx пополам, а команда pop ecx объединит половинку регистра ecx, которая еще хранится в стеке, с регистром ax и всю эту адскую смесь запишет в регистр ecx.

Чтобы понять до конца, как работает стек, исследуем с помощью отладчика программу, которая пытается поменять содержимое регистров ax и ecx (листинг 3.3).

Листинг 3.3. Попытка регистров ax и eax обменяться содержимым

```
.386
.model  flat.stdcall
.code
start:
mov ax.  2211h
```

продолжение ⇨

Листинг 3.3 *(продолжение)*

```
mov ecx, 66554433h
push ax             ;esp=esp-2
push ecx            ;esp=esp-4
pop  ax             ;esp=esp+2, ax=4433h
pop  ecx            ;esp=esp+4, ecx=22116655h
ret
end start
```

Состояние программы перед исполнением инструкции push ax показано на рис. 3.2.

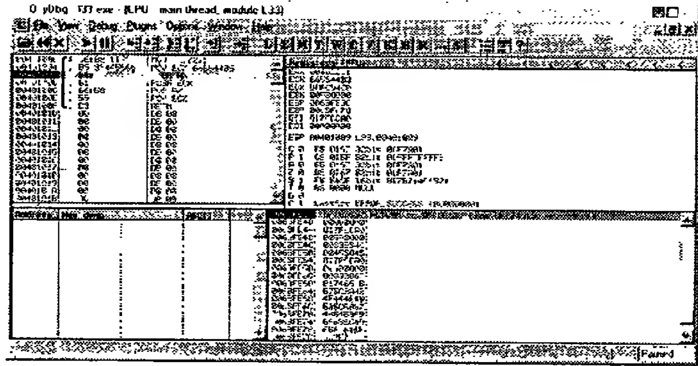


Рис. 3.2. Программа перед первой инструкцией push

В правом нижнем окне отладчика, которое мы до сих пор не замечали, видно состояние стека. Серой полосой выделена *вершина* стека, то есть те байты, которые первыми будут забраны из стека командой pop. У вершины стека есть адрес, который процессор хранит в регистре esp[1]. В правом верхнем окне видно, что esp содержит число 0063fe3c. Это же число выделено серым цветом в правом нижнем окне. Остается только понять, адрес *какого* байта хранит esp.

[1] Регистр esp отладчик показывает следом за уже известными нам регистрами eax, ebx, ecx, edx в своем правом верхнем окне.

Если бы речь шла об обычных данных, показываемых в левом нижнем окне отладчика, то адрес 0063fe3c относился бы к байту BF — первому справа от адреса (рис. 3.3). Но в стеке, как мы скоро увидим, все происходит с точностью до наоборот, и адрес вершины относится к самому дальнему байту 37 (на рис. 3.3 выделен белым).

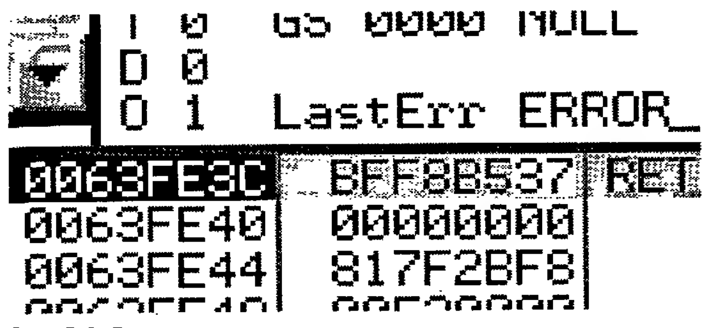


Рис. 3.3. Вершина стека под микроскопом

Как меняется стек после команд push и pop (стрелками отмечены адреса его вершины), показано на рис. 3.4.

```
                    37 B5 F8 BF
                    ↑ esp = 0063fe3c

              11 22 37 B5 F8 BF              push ax
              ↑ esp = 0063fe3a

  33 44 55 66 11 22 37 B5 F8 BF              push ecx
  ↑ esp = 0063fe36

  33 44 55 66 11 22 37 B5 F8 BF              pop ax, ax = 4433
        ↑ esp = 0063fe38

  33 44 55 66 11 22 37 B5 F8 BF              pop ecx, ecx = 22116655
                    ↑ esp = 0063fe3c
```

Рис. 3.4. Состояния стека после команд push и pop

Прежде всего, заметим, что стек растет в сторону *уменьшения* адресов. Действительно, после команды push ax в стеке прибавляется два байта, в то время как адрес вершины уменьшается на два и становится равным 0063fe3a. Регистр ax, равный 2211, помещается в стеке так, что старший байт 22 имеет старший же адрес — как и положено процессорам Intel (см. раздел «Адреса»).

Следующая команда push ecx имеет дело с 4-байтовым регистром, поэтому вершина стека уменьшается на 4 и становится равной 0063fe36. Сам же регистр ecx выворачивается в стеке наизнанку по правилу: чем старше байт, тем старше адрес.

Если команда push уменьшает адрес вершины, то нет ничего удивительного в том, что противоположная команда pop ax увеличивает этот адрес ровно на число доставаемых из стека байтов (в нашем случае это 2). Забираются байты, ближайшие к вершине, в нашем случае это 33 и 44. Поэтому после команды pop ax в регистре ax окажется число 4433 (еще раз вспомним, что адрес старшего байта для процессоров Intel всегда больше). Следующая команда pop ecx заберет из стека оставшиеся четыре байта, после чего адрес вершины увеличится на 4 и достигнет первобытного состояния, какое у него было до выполнения программы. При этом в регистре ecx, как это видно из рисунка, окажется число 22116655.

Очень важно понимать, что команда pop увеличивает адрес вершины стека, *но не стирает* сами числа. После инструкции pop они по-прежнему лежат в стеке и будут находиться там до следующей команды push, которая их окончательно уничтожит. Числа, вытолкнутые из стека, помечены на рис. 3.4 серым цветом, показывающим, что они никуда не делись и, пока не было

команды push, их еще можно оттуда достать. Как это сделать, обсудим в следующем разделе.

А в этом нам осталось подвести итог: программа из листинга 3.3 не справилась со своей задачей. Призванная поменять содержимое регистров, она их безнадежно перепутала. Зато она устранила путаницу в головах, ясно показав, что такое стек и как его правильно использовать.

Косвенная адресация

В прошлом разделе мы поняли, что данные, извлеченные из стека, сохраняются в памяти. Но как их оттуда достать? Очевидно, команда pop здесь не годится, потому что вершина стека уже «уехала вправо», в сторону увеличения адресов и теперь с ней связаны совсем другие числа.

Раньше (см. раздел «Адреса») мы использовали метки для доступа к данным, но память, занимаемая стеком, лишена меток. Единственный ориентир в ней — его вершина. Поэтому для доступа к уже вытолкнутым из стека числам приходится использовать так называемую *косвенную адресацию*, когда адрес участка памяти указывается в одном из регистров процессора.

Предположим, что нам нужно прочитать байт, находящийся на вершине стека. Воспользоваться командой pop тут нельзя, потому что вытолкнуть можно только слово или двойное слово, но никак не байт. Поможет нам указатель стека esp, где как раз и хранится адрес этого байта. Команда процессора, читающая байт, адрес которого записан в регистре esp, выглядит так:

```
mov bl, [esp]
```

Квадратные скобки, окружающие регистр, здесь необходимы, потому что команда mov bl, esp означает попытку послать содержимое регистра esp в регистр bl. Такая команда бессмысленна, и ассемблер не примет ее, выдав сообщение об ошибке, потому что нельзя уместить четыре байта (таков размер регистра esp) в одном.

Теперь можно вернуться к задаче, поставленной в начале этого раздела, и попытаться прочитать числа, оставшиеся в стеке после выполнения двух команд pop. Как показывает рис. 3.3, 2-байтовое слово 2211, первоначально хранимое в регистре ax, находится в двух байтах от вершины стека, а число 66554433, покинувшее регистр ecx, — в шести. Чтобы восстановить значения ax и ecx, нужны такие команды процессора:

```
mov ax, [esp-2]
mov ecx,[esp-6]
```

Выполняя первую из них, процессор обратится к памяти, адрес которой на 2 меньше того, что записан в регистре esp, возьмет оттуда два байта (их адреса будут равны esp-2 и esp-1) и запишет их в регистр ax. Вторая команда выполнится аналогично, только число байтов и адрес будут другими. Программа, «подбирающая» оставленные в стеке числа, показана в листинге 3.4.

Листинг 3.4. Пример косвенной адресации

```
.386
.model  flat,stdcall
.code
start:
mov ax,  2211h
mov ecx,66554433h
mov bl,[esp]; bl=37h
push ax
push ecx
pop  ax
pop  ecx
mov  ecx,[esp-6]; ecx=66554433h
mov  ax, [esp-2]; ax=2211h
ret
end start
```


Заметим, что косвенная адресация не меняет содержимое регистра, хранящего адрес памяти. После выполнения инструкции mov ecx, [esp-6] адрес, хранящийся в esp, останется прежним.

Кроме esp в косвенной адресации могут участвовать и другие регистры: eax, ebx, ecx, edx, ebp, esi, edi. Регистры ebp, esi, edi, до сих пор нам незнакомые, можно поделить только на две части. У регистра ebp есть 2-байтовая «половинка» bp, но регистр bp уже неделим. То же самое относится и к регистрам esi и edi, у которых есть «половинки» si и di, но нет «четвертинок». Есть половинка sp и у регистра esp, но ее содержимое вряд ли интересно, поскольку sp хранит лишь часть адреса вершины стека. Регистр esp стоит особняком в ряду других регистров процессора, потому что у него особая роль — следить за вершиной стека.

Процедуры

Устройство стека, с которым мы только что познакомились, кажется весьма странным, и не очень понятно, зачем нужна память, из которой первым извлекается то, что вошло последним. Пример с обменом чисел не очень убеждает, потому что в ассемблере существует специальная команда xchg, которая меняет содержимое регистров, например, команда xchg eax,ecx делает то же, что и фрагмент программы

```
push ecx
push eax
pop  ecx
pop  eax,
```

но выглядит гораздо яснее и короче.

Понять, зачем нужен стек, невозможно без знакомства с таким важнейшим понятием, как *процедура* — обособленная часть программы, выполняющая какую-то определенную задачу. Процедуры нужны, чтобы понизить сложность программы, сделать ее понятной и управляемой.

Не нужно особо напрягать воображение, чтобы представить себе, что в программе, состоящей из тысяч строк, есть ошибка. Но поистине кошмарной должна быть фантазия, чтобы представить себе, как искать ошибку в программе, которая состоит из «одного куска» и не содержит никаких обособленных частей.

Поэтому опытные программисты стараются составить большие программы из отдельных независимых модулей — процедур. И тогда поиск ошибки значительно упрощается, потому что появляется возможность сначала определить ошибочную процедуру, а затем искать ошибку уже в ней, а не во всей программе. Если какая-то процедура становится слишком большой, можно разбить ее на несколько меньших, словом, писать программу так, чтобы сложность составляющих ее частей была постоянной.

Простейшая процедура AddDigs, складывающая два числа, показана в листинге 3.5.

Листинг 3.5. Процедура AddDigs

```
.386
.model  flat.stdcall
option casemap:none
.code
start:
mov ax.2
mov bx.3
call AddDigs
ret
AddDigs proc
add ax.bx
ret
AddDigs endp
end start
```


Чтобы в имени процедуры AddDigs различались строчные и прописные буквы, программа использует директиву option casemap:none, без которой имена ADDDIGS, adddigs, AddDigs будут для ассемблера одинаковыми.

Сама процедура задается следующим образом:

```
AddDigs proc
add ax.bx
ret
AddDigs endp
```

В ней можно выделить заголовок AddDigs proc, состоящий из имени процедуры и слова proc, признак конца процедуры, состоящий из имени и слова endp, и «тело» процедуры, то есть выполняемые ею инструкции. Процедура вызывается инструкцией call <имя> (в нашем случае это call AddDigs). После вызова выполняются инструкции, из которых состоит процедура, и затем процессор переходит к инструкциям, следующим непосредственно за вызовом процедуры.

Наша «игрушечная» процедура выполняет всего две инструкции: add ax.bx (сумма оказывается в регистре ax) и ret, но этого достаточно, чтобы понять, как процессор умудряется продолжить работу, начиная с инструкции, непосредственно следующей за вызовом call <имя процедуры>.

Все дело в том, что адрес, куда нужно вернуться после выполнения процедуры (адрес возврата), процессор запоминает в стеке. То есть команда call помещает адрес следующей инструкции в стек, затем переводит про-

цессор к адресу первой инструкции процедуры. Эти инструкции выполняются до тех пор, пока не встретится инструкция ret, которая достает из стека адрес возврата, предъявляет его процессору, и тот как ни в чем не бывало начинает выполнять инструкции с этого адреса.

Чтобы всегда знать, чем заняться, процессор имеет специальный регистр eip, содержащий адрес текущей команды. Команда call запоминает в стеке адрес возврата и загружает в eip адрес первой инструкции процедуры. Когда выполняются инструкции процедуры, eip меняется автоматически — в зависимости от самих инструкций. Когда же процессор доходит до команды ret, из стека берется адрес возврата, загружается в eip — и процессор послушно, как слепой за поводырем, следует по указанному адресу.

Но при чем здесь стек — спросите вы? Ведь адрес возврата можно хранить в любом месте памяти. Команда call может записывать туда этот адрес, а команда ret — загружать его в eip. И действительно, при вызове одной процедуры стек не нужен. Но представим себе, что одна процедура вызывает другую. В этом случае стек сначала сохранит адрес возврата в основную программу, процессор перейдет к выполнению процедуры и будет заниматься этим, пока не встретит инструкцию call, после чего запомнит в стеке адрес возврата в процедуру и снова перейдет к командам, расположенным уже в другом участке памяти. Наткнувшись на команду ret, процессор снимет с вершины стека адрес возврата в вызвавшую процедуру, затем снова наткнется на ret, вернет из стека адрес возврата в основную программу и, если не будет других вызовов, там и закончит свою работу. Процесс вызова процедур и возврата из них показан на рис. 3.5.

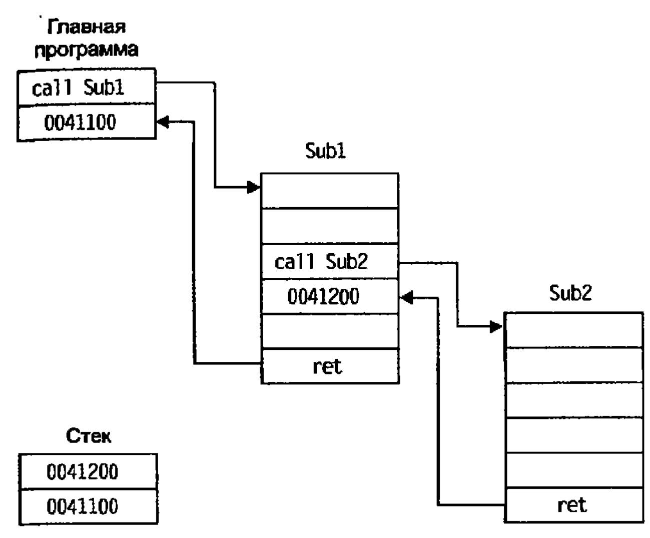


Рис. 3.5. Использование стека при вызове процедур

На рисунке стек показан не в виде линейного участка памяти, а в виде стопки, куда складываются адреса. Как и раньше, стек растет в сторону уменьшения адресов. Команда call кладет адрес возврата в стек и уменьшает на 4 регистр esp. Получается, что новый адрес возврата оказывается каждый раз на вершине стека.

Ясно, что вызываемых процедур может быть сколько угодно: Sub2, показанная на рисунке, может вызвать процедуру Sub3, а та в свою очередь Sub4 и т. д. Команды ret, расположенные в каждой процедуре, найдут на вершине стека правильный адрес возврата, и цепочка вызовов неизбежно закончится в основной программе.

Кроме хранения адресов возврата стек легко можно приспособить для передачи параметров процедуры, хранящих необходимые для ее работы сведения. Параметры нашей первой процедуры AddDigs (см. листинг 3.5) передавались в регистрах ax и bx. Когда их всего два, это

допустимо. Но если параметров десять и более — регистров может не хватить. Поэтому в ассемблере параметры часто заталкиваются в стек командами push, а затем только выполняется команда call. Оказывается, параметры очень легко в этом случае найти в стеке, нужно только знать их число, размер и порядок следования.

Чтобы понять, как параметры передаются через стек, изучим программу, показанную в листинге 3.6.

Листинг 3.6. Передача параметров через стек

```
.386
.model  flat,stdcall
option casemap:none
.code
start:
push dword ptr 2
push dword ptr 3
call AddDigs
ret
AddDigs proc
mov eax,[esp+8] ; eax=2
add eax,[esp+4] ; eax=5
ret 8
AddDigs endp
end start
```

Команда push dword ptr 2 записывает в стек двойное слово (4 байта), содержащее число 2. Ассемблер MASM записал бы двойное слово и по команде push 2, но надежнее указывать размер числа явно[1].

Чтобы правильно написать подпрограмму, необходимо отчетливо представить себе, что находится в стеке после двух команд push и команды call AddDigs.

Очевидно, в стеке хранятся три 4-байтовых числа, первым (его адрес[2] наибольший) идет число 00000002,

[1] Такие маленькие числа, как в нашем примере, можно было бы передать стеку и в обычном 2-байтовом слове. С этим справилась бы команда push word ptr 2.

[2] Строго говоря, адрес может быть только у байта. Под адресом числа понимается адрес его младшего байта.

затем число 00000003 и, наконец, на вершине стека находится адрес возврата (рис. 3.6).

```
0f104000   03000000   02000000
↑esp       ↑esp+4     ↑esp+8


           0040100f ←esp
           00000003 ←esp+4
           00000002 ←esp+8
```

Рис. 3.6. Использование стека для передачи параметров

В верхней части рисунка показано, как расположены числа в памяти компьютера — байт за байтом. Наименьший адрес у 0f — младшего байта адреса возврата. Этот адрес хранится в регистре esp, потому что адрес возврата находится на вершине стека. Вслед за ним в сторону увеличения адреса идут параметры процедуры — 8-байтовые числа 00000003 и 00000002. В стеке они, как и любые другие числа, вывернуты наизнанку: у младшего байта меньший адрес.

Очень часто стек изображают в виде стопки чисел, как это показано в нижней части рис. 3.6. Такой способ не отражает действительного положения байтов в памяти, зато он позволяет яснее увидеть расположенные в стеке числа и понять, как добраться до параметров, переданных процедуре. Очевидно, число 2, помещенное в стек первым, отстоит от вершины на 8 байтов, а число 3 — на 4. Поэтому параметры процедуры будут иметь адреса esp+4 и esp+8. Используя косвенную адресацию, получим две инструкции, складывающие числа:

```
mov eax,[esp+8] ; eax=2
add eax,[esp+4] ; eax=5
```

Их результат — число 5 в регистре eax, можно использовать в основной программе, куда мы возвращаемся с помощью инструкции ret 8. Наверное, вы догадываетесь, что эта восьмерка связана с переданными процедуре параметрами. И это действительно так. Если бы мы просто написали ret, процессор снял бы с вершины стека адрес возврата, и мы благополучно вернулись бы в основную программу. Но при этом в стеке остался бы «мусор» — два переданных параметра. Чтобы освободить от них стек, инструкция ret 8 берет оттуда адрес возврата и затем увеличивает на 8 указатель стека. В результате esp увеличивается на 12 (на 4 — при получении адреса возврата и на 8 после выполнения инструкции ret 8) и возвращается в то состояние, которое было до вызова процедуры.

Не могу молчать

До сих пор у наших программ не было связи с внешним миром. Замкнутые в себе, погруженные во тьму и безмолвие, они не могли ни прочитать что-то с клавиатуры, ни вывести результаты своей работы на экран монитора. О том, что творилось у них внутри, мы узнавали с помощью отладчика.

Настало время переселить «душу» программы в «тело» компьютера, чтобы она получила доступ к монитору, клавиатуре, жесткому диску, звуковой плате и т. д.

Обычно программам помогает общаться с окружающим миром операционная система, которая берет на себя все детали взаимодействия с внешними устройствами. Делается это в разных системах по-разному, в системе Windows этому служат процедуры Windows API, которые вызываются так же, как и любая другая

процедура ассемблера — инструкцией call. Параметры, необходимые этим функциям, передаются через стек.

Прежде чем пытаться вывести что-то на экран, рассмотрим процедуру попроще. Это ExitProcess — процедура, которую вызывает каждая Windows-программа, чтобы завершить свою работу. До сих пор вместо ExitProcess мы пользовались простой инструкцией возврата ret. Но ExitProcess действует гораздо правильней, не только возвращая управление операционной системе, но и освобождая занятые программой ресурсы.

В листинге 3.7 показана программа для Windows, которая только и умеет делать, что правильно завершаться.

Листинг 3.7. Программа, которая умеет правильно завершаться

```
.386
.model  flat,stdcall
option casemap:none
includelib \myasm\lib\kernel32.lib
ExitProcess proto :DWORD
.code
start:
push 0
call ExitProcess
end  start
```

Вызываемая в ней процедура ExitProcess требует одного параметра это код завершения, возвращаемый операционной системе. Он передается процедуре командой push 0. Число 0 считается признаком удачного завершения. Естественно, код завершения в зависимости от обстоятельств может быть иным.

Поскольку ExitProcess — «чужая» процедура, не определенная в нашей программе, ассемблер должен знать, где она находится, а также (для проверки — она ли это) число и размер ее параметров.

Сведения об адресе и параметрах процедуры хранятся в файле библиотеки kernel32.lib, который подключается к ассемблерному тексту директивой

```
includelib \myasm\lib\kernel32.lib
```

Перед тем как создать инструкцию вызова этой процедуры компоновщик (см. раздел «Создание программы» главы 1) сравнивает сведения из библиотеки с *прототипом* ExitProcess proto :DWORD, и если все совпадает, создает пригодный к исполнению файл с расширением .exe. Прототип процедуры очень прост и состоит из имени, слова proto и параметров. В нашем случае параметр один — это двойное слово (то есть, 4 байта) DWORD. Если параметров у процедуры несколько, они разделяются в списке запятой.

Команды push <параметр> и вызовы процедуры call можно в ассемблере фирмы Microsoft сокращенно записать как invoke <имя>. параметр. параметр... Минимальная программа, использующая вызов процедуры invoke, показана в листинге 3.8.

Листинг 3.8. Программа, использующая вызов процедуры invoke

```
.386
.model  flat.stdcall
option casemap:none
includelib \myasm\lib\kernel32.lib
ExitProcess proto :DWORD
.code
start:
invoke ExitProcess. 0
end  start
```

Нужно понимать, что invoke — не команда процессора, а лишь удобная запись вызова процедуры, которая будет преобразована ассемблером в команды push (их будет столько, сколько параметров у процедуры) и завершающий call.

После длинного вступления мы, наконец, готовы написать программу, выводящую на экран слова «Не могу молчать!». Ее текст показан в листинге 3.9.

Листинг 3.9. Первые слова

```
.386
.model  flat. stdcall
option  casemap:none
ExitProcess  proto :dword
GetStdHandle proto :dword
WriteConsoleA   proto :dword.:dword.\
               :dword.:dword.:dword
includelib  \myasm\lib\kernel32.lib
.data
stdout   dd ?
msg      db «Не могу молчать!».0dh.0ah
cWritten dd ?
.code
start:
invoke  GetStdHandle. -11
mov stdout. eax
invoke  WriteConsoleA. stdout. ADDR msg.\
        sizeof msg. ADDR   cWritten. 0
invoke  ExitProcess. 0
end start
```

В программе вызываются две новые процедуры: GetStdHandle и WriteConsoleA. Их прототипы приводятся в начале программы. Прототип процедуры WriteConsoleA не уместился на одной строке. Чтобы показать, что описание процедуры будет продолжено на следующей строке, используется косая черта \. Та же черта видна и в строке, где вызывается WriteConsoleA. На этот раз она показывает, что в одной строке не уместился список реальных параметров процедуры[1].

Процедура GetStdHandle, как можно догадаться по ее названию, получает *дескриптор стандартного устройства* — число, которое нужно указывать другим процедурам, взаимодействующим с этим устройством. Единственный параметр этой процедуры показывает, *какого*

[1] Кроме косой черты признаком продолжения строки служит и запятая. Можно так построить вызов процедуры, что косая черта не понадобится.

рода дескриптор нужно получить. Чтобы, например, узнать дескриптор стандартного устройства вывода, куда будет отправлена фраза «Не могу молчать», параметр должен быть равен −11. Как и многие другие процедуры, GetStdHandle помещает результат своей работы в регистр eax. Поэтому нужна еще одна инструкция mov stdout, eax, чтобы сохранить полученный дескриптор в памяти.

Процедура WriteConsoleA, выводящая символы на экран, выглядит гораздо сложнее, у нее пять параметров, хотя последний, пятый, никакого смысла не имеет и всегда равен нулю. Первые четыре параметра таковы:

1. stdout — это дескриптор стандартного устройства вывода (экрана монитора), полученный процедурой GetStdHandle.
2. ADDR msg — адрес начала сообщения. Чтобы получить его, используется специальный оператор получения адреса ADDR. Как и все в компьютере, сообщения представлены последовательностями чисел. Каждая буква сообщения кодируется определенным числом. Так, например, прописная русская буква Н кодируется[1] числом $8D_{16}$ или 141_{10}. Поскольку букв не так много, достаточно 256 чисел, чтобы закодировать два любых алфавита (например, латинский и кириллицу). Поэтому один символ хранится в одном байте. Кроме букв есть еще невидимые служебные символы перевода строки, пробела, табуляции, которые также умещаются в одном байте.

[1] Есть несколько соглашений о том, каким числом какую букву кодировать. Такие соглашения называются кодировками. Число 14110 представляет символ 'Н' в альтернативной кодировке. Кроме альтернативной, распространена еще и кодировка Windows, в которой буква 'Н' представлена числом 205_{10}.

3. SIZEOF msg — размер сообщения, то есть число байтов в нем. Наше сообщение, заключенное в кавычки, состоит из 18 байтов (16 байтов занимают буквы и два байта — символы 0dh, 0ah[1], которые командуют процедуре WriteConsoleA перевести строку[2]). Размер сообщения (число байтов от указанной метки msg до следующей cWritten) программа-ассемблер вычисляет во время компиляции.
4. ADDR cWritten — адрес участка памяти, где процедура WriteConsoleA сохранит число выведенных на экран символов.

Разбор полетов

Предыдущий раздел оказался очень трудным из-за того, что вызов процедур Windows API требует знания многочисленных параметров и операторов языка. И вряд ли он может быть существенно улучшен. Можно только пересказать его другими словами, что мы сейчас и сделаем.

Итак, попробуем проследить за программой, показанной в листинге 3.9, с помощью отладчика. На рис. 3.7 видны команды процессора и область данных, созданные ассемблером. Строка invoke GetStdHandle, -11 листинга 3.9 заменяется ассемблером на команду push -0B и вызов процедуры call. Нажав дважды клавишу F8, увидим в регистре eax число 12_{10} ($0C_{16}$). Это и есть дескриптор стандартного устройства вывода. Дескриптор, как и все в компьютере, — всего лишь число и ничем другим быть не может.

Следующая команда mov stdout, eax, показанная отладчиком как MOV DWORD PTR DS:[403000], EAX, посылает

[1] Байты 0dh, 0ah записаны в шестнадцатеричной системе. В десятичной системе они равны 13 и 10 соответственно.

[2] Кавычки, обрамляющие фразу, не учитываются и на экране не отображаются.

содержимое eax в ячейку памяти с адресом 00403000. Слова dword ptr говорят о том, что в ячейке 4 байта, квадратные скобки, окружающие число 403000, показывают нам, что это адрес, а значок DS: изображает так называемый сегментный регистр, который в плоской (model flat) модели памяти никакой роли не играет, поэтому программисту нечего о нем думать[1].

Говоря об адресе ячейки, мы, как всегда, имеем в виду адрес ее младшего байта. Всего в ячейке 4 байта, на рис. 3.7 они видны в самом начале области данных, перед символами «Не могу молчать». После команды MOV DWORD PTR DS:[403000]. EAX там окажется число 0C000000, то есть вывернутое наизнанку 0000000C или 12_{10}.

```
00401000  $ 6A F5          PUSH -0B
00401002  . E8 2B000000    CALL <JMP.&kernel32.GetStdHandle>
00401007  . A3 00304000    MOV DWORD PTR DS:[403000],EAX
0040100C  . 6A 00          PUSH 0
0040100E  . 68 16304000    PUSH L39.00403016
00401013  . 6A 12          PUSH 12
00401015  . 68 04304000    PUSH L39.00403004
0040101A  . FF35 00304000  PUSH DWORD PTR DS:[403000]
00401020  . E8 13000000    CALL <JMP.&kernel32.WriteConsoleA>
00401025  . 6A 00          PUSH 0
00401027  . E8 00000000    CALL <JMP.&kernel32.ExitProcess>

Address   Hex dump                  ASCII
00403000  00 00 00 00 8D A5 20 AC   ....Не м
00403008  AE A3 E3 20 AC AE AB E7   огу молч
00403010  A0 E2 EC 21 0D 0A 00 00   ать!....
00403018  00 00 00 00 00 00 00 00   ........
```

Рис. 3.7. Команды процессора и данные в окнах отладчика

За вызовом процедуры GetStdHandle в нашей программе идет нечто более значительное, а именно — вызов WriteConsoleA. Ему предшествует заталкивание в стек многочисленных параметров этой процедуры. Причем, заметьте, первым в стек отправляется последний, пятый по счету параметр, то есть ноль. Именно такой порядок, задаваемый директивой .model flat. stdcall, принят для процедур Windows API. Кроме того, слово «stdcall» в задании модели памяти значит, что процедура должна сама заботиться о восстановлении стека, она должна «убирать за собой», чтобы стек оказался в том же состоянии, что и до вызова.

[1] Сегментные регистры CS, DS, SS, ES, GS, FS участвуют в формировании адреса, но в Windows их правильные значения устанавливает операционная система. В других моделях памяти, о которых мы будем еще говорить, программист должен сам менять сегментные регистры и указывать их в командах ассемблера.

Но вернемся к нашим параметрам. Второе число, попавшее в стек перед вызовом WriteConsole, — это 00403016 (в тексте программы на ассемблере оно записывается как ADDR cWritten; отладчик отображает команду довольно замысловато: PUSH L39.00403016, но если ее выполнить клавишей F8, в стеке окажется число 00403016 (обязательно убедитесь в этом). 00403016 — адрес ячейки памяти, куда WriteConsole запишет количество показанных на экране символов. Как видно из рис. 3.7, эта ячейка идет следом за символами Не могу молчать!.

Третье число ассемблер получит, применив оператор SIZEOF к метке msg. Как следует из рисунка, число это, равное 12_{16} или 18_{10}, отправляется в стек командой push 12.

Следующий параметр — адрес начала последовательности символов, выводимых на экран. В исходном тексте программы он показан как ADDR msg. Ассемблер вычисляет этот адрес во время компиляции программы, а процессор видит перед собой лишь скупую, неумолимую команду: поместить в стек 00403004 (отладчик показывает ее как PUSH L39.00403004). Процессор выполняет то, что приказано, ничего не зная о числе, — адрес ли это, переменная или что-то еще.

И наконец, последнее обращение к стеку выглядит в окне отладчика так:

```
PUSH DWORD PTR DS:[403000]
```

По этой команде процессор помещает в стек дескриптор стандартного устройства вывода, хранящийся

в 4 байтах памяти, начиная с адреса 00403000. В исходном тексте программы он помечен как stdout.

Только что команды ассемблера предстали перед нами в неглиже — такими, какими их видят отладчик и процессор. Чтобы не запутаться, взглянем еще раз на исходный текст программы из листинга 3.9, записанный чуть иначе (листинг 3.10).

Листинг 3.10. Использование готовых прототипов процедур

```
.386
.model  flat, stdcall
option  casemap:none
include    \myasm\include\windows.inc
include    \myasm\include\kernel32.inc
includelib \myasm\lib\kernel32.lib
.data
stdout    dd ?
msg       db «Не могу молчать!»,0dh,0ah
cWritten  dd ?
.code
start:
invoke  GetStdHandle, STD_OUTPUT_HANDLE
mov stdout, eax
invoke  WriteConsoleA, stdout, ADDR msg, \
        sizeof msg, ADDR   cWritten, NULL
invoke  ExitProcess, 0
end start
```

Можно подумать, что в этой программе нет прототипов процедур, на самом же деле все они просто переселились в подключаемый файл kernel32.inc[1]. А огромный файл windows.inc содержит всего одну полезную нашей программе строчку STD_OUTPUT_HANDLE equ -11, говорящую ассемблеру, что все имена STD_OUTPUT_HANDLE, встреченные в программе, нужно заменить на –11. Такие строки часто применяются в программах на ассемблере, потому что символические имена гораздо понятней, чем просто числа.

[1] Подключаемые файлы .inc и .lib — это часть компилятора. Они, как это видно из листинга 3, 10, находятся в папках \myasm\include\ и \myasm\lib\.

Своеволие ассемблера

В программах этой главы вызывались как процедуры Windows API, так и единственная самостоятельно написанная нами процедура AddDigs (см. листинг 3.6). Правда, AddDigs мы пока не научились использовать так, как стандартные процедуры — с прототипом и директивой invoke.

Попробуем поэтому привести процедуру AddDigs к общему стандарту и вызвать ее так же, как процедуру Windows. Для этого нужен прототип для AddDigs и заново написанный заголовок процедуры, в котором указываются ее параметры. Программа, использующая преображенную процедуру для сложения двух чисел, показана в листинге 3.11.

Листинг 3.11. Вызов AddDigs с помощью директивы invoke

```
.386
.model  flat,stdcall
option casemap:none
includelib \myasm\lib\kernel32.lib
ExitProcess proto :DWORD
AddDigs proto :DWORD, :DWORD
.code
start:
invoke AddDigs,2,3
invoke ExitProcess,0
AddDigs proc arg1:DWORD,arg2:DWORD
mov eax,[esp+8] ; eax=2
add eax,[esp+12]; eax=5
ret
AddDigs endp
end start
```

Задача 3.1. Чем программа из листинга 3.11 отличается от программы из листинга 3.6?

Теперь AddDigs вызывается так же, как и ExitProcess. Но — обратите внимание — параметры теперь снимаются с других полочек стека. Что-то отодвинуло их от вер-

шины, и теперь первое число (двойка) отстоит от вершины на 8 байт (было 4), а второе — на 12 (было на 8).

Что же случилось? Ответ, как обычно, дает отладчик, который обнаруживает в созданной нами процедуре кучу посторонних и на первый взгляд кажущихся непонятными инструкций:

```
PUSH EBP
MOV EBP.ESP
MOV EAX.DWORD PTR SS:[ESP+8]
ADD EAX.DWORD PTR SS:[ESP+C]
LEAVE
RETN 8
```

Но если выяснить, что LEAVE эквивалентна паре инструкций

```
mov esp. ebp
pop ebp.
```

то в своеволии ассемблера начинает угадываться какой-то смысл. Заключая инструкции процедуры в рамку

```
push ebp
mov ebp.esp
...
mov esp.ebp
pop ebp.
```

ассемблер сохраняет указатель стека в регистре ebp. Если ebp не меняется внутри процедуры, то esp можно восстановить перед выходом из нее. Чтобы при этом сохранить ebp для внешнего мира, его сначала отправляют в стек, а перед возвратом из процедуры снова вынимают оттуда. Вот из-за того, что ebp сохраняется в стеке, и меняется положение параметров процедуры. Первым в стек отправляется число 3, затем 2, затем ebp, затем адрес возврата. Значит, ebp отстоит на 4 байта от вершины, число 2 — на 8, а число 3 — на 12 байт.

Теперь нам предстоит понять, зачем esp хранится в регистре ebp, когда в нашей процедуре он вообще не меняется? Затем, что ассемблер не знает, меняется ли

указатель стека, и сохраняет его так, на всякий случай, зная, что стек часто используется для хранения *локальных* переменных.

Локальные переменные необходимы процедуре только в момент ее выполнения, вот почему для них жалко использовать место в компьютерной памяти, выделенное директивой .data. Но если хранить их в стеке, они возникнут при входе в процедуру и исчезнут при выходе из нее.

В программе из листинга 3.12 показано, как заводятся локальные переменные в процедуре StrDisp, выводящей на экран строку символов.

Листинг 3.12. Использование процедурой локальных переменных

```
.386
.model  flat. stdcall
option  casemap:none
include    \myasm\include\windows.inc
include    \myasm\include\kernel32.inc
includelib \myasm\lib\kernel32.lib
StrDisp    proto :DWORD. :DWORD
.data
msg db«Не могу молчать!».0dh.0ah
.code
start:
invoke  StrDisp. ADDR msg.sizeof msg
invoke  ExitProcess. 0
StrDisp proc StrAddr:DWORD. StrSz:DWORD
sub esp.8 ;место локальных переменных
invoke  GetStdHandle. STD_OUTPUT_HANDLE
mov [ebp-4]. eax
invoke  WriteConsoleA. [ebp-4]. [ebp+8]. \
        [ebp+12]. ADDR [ebp-8]. NULL
ret 8 ;освободить стек от параметров
StrDisp endp
end start
```

Процедура StrDisp упрощает вывод символов на экран, ведь ей необходимы всего два параметра — адрес начала строки и число символов в ней. Она служит как

бы «оберткой» для GetStdHandle и WriteConsoleA, скрывая внутри себя такие служебные переменные, как дескриптор экрана и число отображенных символов.

Эти переменные разумно сделать локальными, чтобы они жили только в момент выполнения процедуры, а при выходе из нее пропадали. Проще всего разместить такие переменные в стеке, уменьшив esp на число занимаемых ими байтов. Когда произойдет выход из процедуры, стек придет в первобытное состояние, а локальные переменные просто «смоет волной». В нашем случае переменных две, каждая из них занимает 4 байта, следовательно, из esp нужно вычесть 8, что и делает инструкция sub esp, 8.

Уменьшать указатель стека на суммарный размер локальных переменных нужно потому, что иначе эти переменные могут быть уничтожены вызовом из текущей процедуры каких-либо еще процедур. Ведь каждый вызов связан с заталкиванием в стек параметров и адреса возврата, которые, если не уменьшить esp, уничтожат локальные переменные. Но если вычесть из esp суммарный размер локальных переменных, те попадут в «мертвую» зону, куда не проникают ни параметры, ни адреса возврата других процедур, *вызванных внутри нашей*. Ведь все эти параметры и адреса возврата окажутся *выше* наших локальных переменных, если представить стек в виде стопки и учесть, что он растет в сторону уменьшения адресов.

Раз указатель стека может меняться внутри процедуры, адреса параметров и локальных переменных следует отсчитывать относительно ebp. Вспомним, что ebp хранит указатель стека непосредственно *перед* его уменьшением sub esp, 8. Значит, адреса локальных переменных станут *больше* ebp, адреса же параметров — меньше. Адрес первой локальной переменной будет ebp-4, второй — ebp-8.

Расстояние параметров от точки, на которую указывает ebp, будет таким же, как и в процедуре AddDigs из

листинга 3.11. Длина выводимой на экран строки заталкивается в стек первой и потому находится дальше всех от точки, на которую указывает ebp. Следом идет адрес выводимой на экран строки, затем адрес возврата из процедуры, и, наконец, вершину стека занимает сам сохраненный ebp. Значит, длина строки имеет адрес ebp+12, а ebp+8 — это адрес ее начала. Состояние стека после запуска процедуры и выделения локальных переменных показано на рис. 3.8.

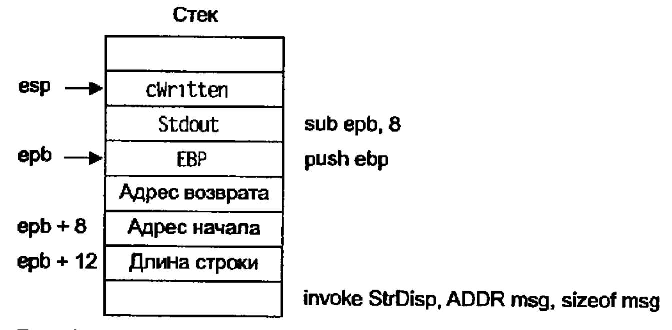


Рис. 3.8. Параметры и локальные переменные процедуры

Теперь в нашей процедуре StrDisp все должно быть понятно, кроме, быть может, странной передачи параметра ADDR [ebp-8] процедуре WriteConsole. Но ничего странного здесь нет. По адресу ebp-8 хранится локальная переменная — число показанных на экране символов. Но если указать процедуре просто [ebp-8], то передастся само число, а не его адрес! Вот почему нужно писать ADDR [ebp - 8]. Теперь в стек отправится *адрес переменной, находящейся по адресу* ebp-8, то есть ассемблер отправит в стек разность ebp и 8.

Высчитывание адресов локальных переменных и параметров довольно утомительно и чревато ошибками. Вот почему ассемблер предлагает другой, более удобный способ обращения с параметрами, переданными проце-

дуре, и локальными переменными. Вместо ручного уменьшения указателя стека можно задать имена локальных переменных директивой LOCAL, а вместо отсчитывания адреса от ebp можно использовать имя параметра, указанное в заголовке процедуры. С учетом этих нововведений наша процедура GetStdHandle окажется такой.

Листинг 3.13. Задание локальных переменных директивой LOCAL

```
StrDisp proc StrAddr:DWORD, StrSz:DWORD
LOCAL stdout,cWritten
invoke  GetStdHandle, STD_OUTPUT_HANDLE
mov stdout, eax
invoke  WriteConsoleA, stdout, StrAddr, \
        StrSz, ADDR cWritten, NULL
ret
StrDisp endp
```

Такая запись полностью скрывает механизм передачи параметров процедуре. Ничего нельзя понять и о способе выделения локальных переменных. Даже инструкция возврата записывается как ret, а не ret 8, потому что ассемблер знает число и размер параметров процедуры и потому не нуждается в подсказке. Но отладчик, конечно, обнаружит все то, что мы уже видели в листинге 3.12: вычитание из esp общего размера локальных переменных, запоминание указателя стека, указание адреса относительно ebp и т. д.

Такая, как в листинге 3.13, запись процедуры полезна, потому что позволяет назвать локальные переменные человеческими именами и не думать о положении параметров в стеке. Но при этом нужно обязательно знать, что на самом деле творится со стеком и со всей программой. Частичное знание противно духу ассемблера. Нужно представлять себе программу до последней инструкции процессора. Только тогда удастся хорошо ее написать и успешно отладить. Вот почему этот раздел начался с «ручной» передачи параметров и ручного же выделения места для локальных переменных.

ГЛАВА 4 Как решать задачу



Вывод чисел

Лучше всего учиться программировать, решая какую-нибудь сложную задачу. Сложность — понятие относительное. Для нас сейчас весьма сложной будет задача нахождения первых десяти простых чисел. Напомню, что простым называется число, которое делится нацело только на себя и единицу. Число 7 — простое, а число 4 — нет, потому что, кроме 1 и 4, делится еще на 2.

Чтобы справиться с любой сложной задачей, нужно разбить ее на несколько простых. Ничто так не отбивает интерес к программированию, как попытка написать сложную программу без подготовки, сразу, одним куском. Мы поступим иначе и будем сначала решать небольшие частные задачи, а потом применим накопленный опыт в большой программе.

И первой нашей задачей будет отображение чисел на экране. Процедура WriteConsoleA, с которой мы познакомились в конце предыдущей главы, не умеет этого делать, потому что создана для вывода на экран последовательностей *символов*, из которых, к примеру, состоит фраза «Не могу молчать!» Но числа — не символы. Число 1, в зависимости от размера хранящей его ячейки памяти, может занимать и 1, и 2, и 4 байта. Выглядеть оно (с учетом обратного порядка байтов в памяти) будет как 01, или как 0100 или как 01000000. В то же время символ '1' определяется стандартной кодировкой как байт, в котором хранится число 49_{10}.

Значит, число сначала нужно преобразовать в последовательность символов, а уж потом выводить эту последовательность на экран процедурой WriteConsoleA. Для такого преобразования в системе Windows есть специальная процедура wsprintf. В отличие от многих других процедур, число параметров wsprintf переменно и зависит от количества преобразуемых чисел. Но первые параметры всегда одни и те же: это адрес буфера, где процедура сохраняет число в виде последовательности символов, адрес форматной строки, указывающей процедуре, какое выполнить преобразование, и, конечно, само преобразуемое число. Программа, выводящая на экран целое число 123456, показана в листинге 4.1.

Листинг 4.1. Вывод на экран числа 123456

```
.386
.model  flat, stdcall
option  casemap:none
include    \myasm\include\windows.inc
include    \myasm\include\user32.inc
include    \myasm\include\kernel32.inc
includelib \myasm\lib\user32.lib
includelib \myasm\lib\kernel32.lib
BSIZE  equ 15
.data
ifmt        db "%d",0
buf       db BSIZE dup(?)
dig       dd 123456
stdout    dd ?
cWritten  dd ?
.code
start:
invoke  GetStdHandle, STD_OUTPUT_HANDLE
mov stdout, eax
invoke  wsprintf, ADDR buf, ADDR ifmt, dig
invoke  WriteConsoleA, stdout, ADDR buf, \
        BSIZE, ADDR cWritten, NULL
invoke  ExitProcess, 0
end start
```

От предыдущих эта программа отличается прежде всего тем, что в ней появились новые подключаемые

файлы user32.inc и user32.lib, хранящие информацию о функции wsprintf. Число ее параметров у нас пока минимально и равно трем. Первый параметр ADRR buf — адрес буфера, куда будет записана последовательность символов. Память для буфера выделяется строкой

```
bufdb BSIZE dup(?),
```

которая резервирует BSIZE идущих подряд байтов. О том, что выделяется несколько байтов памяти, говорит слово dup — сокращенное от английского слова *duplication* (повторение). Вопросительный знак в скобках после dup говорит о том, что значение байтов заранее не определено.

Размер буфера обозначен именем BSIZE, а реальное число, которым ассемблер заменит BSIZE, задается строкой BSIZE equ 15. Определяя размер таким способом, мы решаем две важные задачи: во-первых, вводим вместо малопонятного числа 15 осмысленное имя, говорящее о том, что перед нами размер буфера (buffer size). Во-вторых, до предела упрощаем изменение размера: вместо выискивания всех мест в программе, где он встречается (не все числа 15 могут, к тому же, иметь к этому отношение), достаточно поменять одну строку.

Второй параметр функции wsprintf — ADDR ifmt — это адрес строки формата, задающей тип преобразования. Эта строка состоит из символов и всегда завершается нулевым байтом. Строка "%d",0 задает преобразование одного целого числа в последовательность символов. Строка "%d %d",0 задает преобразование двух чисел. Есть много других преобразований, о которых мы поговорим позже. А пока стоит скомпилировать программу из листинга и проследить за ее работой с помощью отладчика. Здесь нас подстерегает неожиданность. Оказывается, после вызова wsprintf ассемблер самовольно добавил инструкцию add esp, 12. Сделал он это потому,

что процедура wsprintf сама не знает, сколько у нее параметров[1]. Значит, восстановление стека должен взять на себя компилятор. Наша программа загружает в стек три параметра процедуры wsprintf. Чтобы сделать все «как было», нужно убрать из стека эти три параметра, что и делает инструкция add esp,12.

Переходы

При отыскании простых чисел многократно повторяются одни и те же действия: программа подготавливает число для проверки, а затем начинает делить его на все подряд. Число n нужно делить на все числа от 2 до n – 1. Если хотя бы раз остаток от деления равен нулю, число не простое и нужно переходить к следующему.

Ну а если все остатки от деления не равны нулю? Тогда число простое и нужно его где-то сохранить. Иными словами, необходима инструкция, которая меняла бы ход выполнения программы в зависимости от результата проверки.

Эта инструкция, если подумать, только и делает возможными компьютерные вычисления. Не будь ее, чудовищная скорость компьютеров оказалась бы бесполезной, потому что программисту пришлось бы описывать каждое его действие. И ему не хватило бы собственной жизни, чтобы описать десятую долю секунды жизни процессора.

К счастью, инструкции, меняющие ход выполнения программы, существуют, и в огромном количестве — быть может, как раз потому, что суть программирования именно в них. Программа, показанная в листинге 4.2, сообщает, равно или не равно нулю число digit.

---

[1] Когда, например, строка формата задает преобразование двух чисел, общее число параметров будет равно четырем.

Листинг 4.2. Равно ли нулю число digit?

```
.386
.model  flat, stdcall
option  casemap:none
include    \myasm\include\windows.inc
include    \myasm\include\kernel32.inc
includelib \myasm\lib\kernel32.lib
.data
z  db "равно нулю",13,10
zsize dd ($-z)
nz db "не равно нулю",13,10
nzsize dd ($-nz)
digit dd 0
stdout   dd ?
cWritten dd ?
.code
start:
invoke  GetStdHandle, STD_OUTPUT_HANDLE
mov stdout, eax
cmp digit,0
jnz nzero
invoke  WriteConsoleA, stdout, ADDR z, \
        zsize, ADDR cWritten, NULL
jmp exit
nzero:
invoke  WriteConsoleA, stdout, ADDR nz, \
        nzsize, ADDR cWritten, NULL
exit:   invoke  ExitProcess, 0
end start
```

Самая важная инструкция этой программы, да и вообще всего языка ассемблера, — конечно же, jnz nzero: если флаг Z (см. раздел «Конечность» главы 2) опущен, она приказывает процессору перейти к инструкции с меткой nzero. Если же флаг Z поднят, процессор, как ни в чем не бывало, продолжит работу с инструкции, непосредственно следующей за jnz nzero, то есть вызовет процедуру WriteConsoleA, которая покажет на экране сообщение «равно нулю».

Инструкция условного перехода jnz работает в паре с инструкцией сравнения cmp digit,0. Смысл инструкции cmp в том, что из левого операнда digit *как бы* вычитается правый операнд 0. При этом флаги устанавливаются так, как будто вычитание произошло, сами же операнды не меняются.

Другая важнейшая инструкция, встреченная нами в этой программе, велит процессору без каких-либо условий немедленно перейти к указанной метке. Это инструкция безусловного перехода jmp. Представим себе, что число digit равно нулю. Тогда инструкция jnz nzero не сработает, процедура WriteConsoleA покажет на экране сообщение равно нулю, а дальше необходимо обойти второй вызов процедуры WriteConsoleA, иначе на экране возникнет и сообщение не равно нулю. Вот для такого обхода и создана инструкция безусловного перехода jmp, которая направляет процессор к выходу из программы, то есть к запуску процедуры ExitProcess. Как видим, комбинация условного и безусловного перехода позволяет организовать разные «ветви» вычислений в зависимости от результата проверки.

В программе из листинга 4.2 есть еще одно новшество, о котором нужно сказать, прежде чем перейти к следующему разделу. В ней иначе вычисляется длина сообщения. До сих пор мы использовали оператор SIZEOF. Но можно заставить ассемблер вычислять размер по-другому. Для этого сразу за сообщением объявляется переменная, в которой хранится разница между текущим адресом (он обозначается значком $) и адресом начала сообщения. Например, число zsize, заданное строками

```
z  db "равно нулю",13,10
zsize dd ($-z)
```

равно 12, потому что таково расстояние в байтах между метками z и zsize (убедитесь в этом сами). Это расстояние ассемблер вычисляет во время компиляции программы.

Повторение

С помощью условных инструкций можно заставить процессор многократно повторять одни и те же действия. Для этого достаточно проверять условие, и если оно выполняется, отбрасывать процессор на несколько инструкций назад. При этом в повторяемых инструкциях должно быть нечто нарушающее условие возврата, иначе процессор работал бы вечно.

В этом разделе мы познакомимся с инструкцией `loop` `<метка>`, которая способна в немногих строках программы уместить огромный объем работы процессора. Действует она просто: увидев инструкцию `loop`, процессор уменьшает на единицу регистр `cx` и проверяет, не равен ли он нулю. Если `cx = 0`, выполняется следующая после `loop` инструкция. Если нет — процессор переходит к указанной метке.

Программа, показанная в листинге 4.3, выводит на экран 10 идущих подряд чисел от 1 до 10.

Листинг 4.3. Вывод на экран чисел от 1 до 10

```
.386
.model  flat, stdcall
option  casemap:none
include    \myasm\include\windows.inc
include    \myasm\include\user32.inc
include    \myasm\include\kernel32.inc
includelib \myasm\lib\user32.lib
includelib \myasm\lib\kernel32.lib
BSIZE      equ 15
.data
ifmt    db       "%d",0
buf              db BSIZE dup(?)
crlf             db 0dh,0ah
stdout           dd ?
cWritten         dd ?
.code
start:
invoke  GetStdHandle, STD_OUTPUT_HANDLE
mov stdout, eax
mov edx,1
mov ecx,10
nxt:
   push    ecx
   push    edx
   invoke  wsprintf, ADDR buf, ADDR ifmt,edx
   invoke  WriteConsoleA, stdout, ADDR buf,\
     BSIZE, ADDR cWritten, NULL
   invoke  WriteConsoleA, stdout, ADDR crlf,\
   2, ADDR cWritten, NULL
   pop     edx
   inc     edx
   pop     ecx
loop    nxt
invoke  ExitProcess, 0
end start
```


Самое главное в ней — пространство от метки `nxt` до инструкции `loop`, называемое *циклом*. Внутри цикла помещены инструкции, выводящие на экран числа, хранимые в регистре `edx`.

Сначала процедура `wsprintf` преобразует число в последовательность символов, затем процедура `WriteConsoleA` выводит эти символы на экран. Второй вызов `WriteConsoleA` нужен для перевода строки и возврата к левому краю экрана (этим ведают символы 0dh,0ah или в десятичном представлении 13,10).

Перед началом цикла в регистр `ecx` посылается число 10, а в регистре `edx` оказывается единица. Далее оба регистра сохраняются в стеке. Делается это из-за того, что процедуры Windows сами используют эти регистры для своих внутренних нужд, поэтому сказать, что будет с `edx` или с `ecx` после вызова процедуры, нельзя[1].

После сохранения регистров на экран выводится текущее число, равное единице при первом обороте

---

[1] Правда, известно, что процедуры Windows API сохраняют значения регистров ebx, edi, esi и ebp.

цикла. А дальше начинается самое интересное. Сохраненные регистры ecx и edx достаются из стека, регистр edx увеличивается на единицу инструкцией inc edx и становится равным двум. Перед командой loop nxt регистр ecx равен 10. Инструкция loop уменьшает ecx на единицу и проверяет, равен ли ecx нулю. В нашем случае это не так, процессор перейдет к метке nxt и начнется второй оборот цикла.

Очевидно, при cx = 10 цикл будет выполняться 10 раз (при значениях ecx 10, 9, 8, 7, 6, 5, 4, 3, 2, 1). Когда ecx сравняется с нулем, процессор перейдет к инструкции, стоящей после loop. В нашем случае это вызов процедуры ExitProcess.

Заметим, что цикл, организованный с помощью инструкции loop, выполняется по крайней мере один раз. Начальное значение «счетчика цикла» ecx равно при этом 1. Если перед исполнением цикла сделать ecx равным нулю, то инструкция loop вычтет из нуля единицу и в результате получится число 0ffffffff. А это значит, что вместо нуля цикл выполнится 4 294 967 295 раз! В ассемблере есть, конечно, возможность сделать цикл, который может не выполняться совсем, но об этом речь впереди.

А сейчас сделаем важное наблюдение над регистрами. Если до сих пор мы считали их одинаковыми, то лишь из-за поверхностного знакомства с ними. Если присмотреться, у каждого откроется свое лицо. Например, инструкция loop работает только с ecx. На протяжении всей книги мы будем всматриваться в регистры, стараясь изучить их свойства и повадки.

Деление

Для проверки числа на «простоту» нужно перебрать все возможные делители, отличные от единицы и самого

числа, и убедиться в том, что все остатки от деления не равны нулю. Это и скажет нам, что нет ни одного деления нацело, а следовательно, — число простое.

В процессоре Intel остатки от деления — побочный продукт самого деления. Поэтому задача, которую мы сами себе поставили, требует знания инструкции процессора div.

В сущности, буквами div обозначаются несколько различных операций деления. Все зависит от типа аргумента инструкции div, то есть делителя. Если аргументом служит байт, как, например, в инструкции div bl, то процессор поделит число в регистре ax на bl и запишет частное от деления в регистр al, а остаток — в регистр ah.

Если аргумент команды div — слово (например, div bx), то процессор поделит число, старшие биты которого хранит регистр dx, а младшие — ax. После деления частное окажется в регистре ax, а остаток — в регистре dx.

И наконец, если делитель — двойное слово, как в инструкции div ebx, то процессор считает, что делимое хранится в двух двойных словах. Старшие биты делимого он возьмет из edx, младшие — из eax, а после деления частное окажется в eax, а остаток — в edx.

Какую же из трех инструкций выбрать для нашей задачи? Будем стараться исследовать на «простоту» как можно больше чисел, поэтому выберем третью инструкцию, но для простоты пока будем хранить делимое только в eax, а edx пусть будет равен нулю.

Программа, показанная в листинге 4.4, делит 100 на 3 и показывает частное (конечно, это 33) на экране.

Листинг 4.4. Пример деления

```
.386
.model  flat, stdcall
option  casemap:none
include    \myasm\include\windows.inc
```

продолжение

Листинг 4.4 *(продолжение)*

```
include    \myasm\include\user32.inc
include    \myasm\include\kernel32.inc
includelib \myasm\lib\user32.lib
includelib \myasm\lib\kernel32.lib
BSIZE equ 15
.data
ifmt    db      "%d",00
stdout          dd ?
cWritten        dd ?
.data?
buf     db  BSIZE dup(?)
.code
start:
invoke  GetStdHandle, STD_OUTPUT_HANDLE
mov stdout, eax
mov eax,100
mov edx,0    ;edx:eax - делимое
mov ebx,3    ;ebx - делитель
div ebx      ;eax - частное, edx - остаток
invoke  wsprintf, ADDR buf, ADDR ifmt,eax
invoke  WriteConsoleA, stdout, ADDR buf, \
        BSIZE, ADDR cWritten, NULL
invoke  ExitProcess, 0
end start
```

В программе, кроме самой инструкции div, есть еще одно новшество — директива .data?. Вопросительный знак означает, что данные, описанные после директивы, во-первых, не определены, а во-вторых, не занимают место в исполняемом файле с расширением .exe. Представим себе, что BSIZE равен не 15, а 15000. Тогда определение буфера в области .data привело бы к многократному увеличению размеров исполняемого файла, потому что компилятор выделил бы место для буфера прямо в нем. Но если буфер объявлен в области .data?, размер файла .exe не увеличивается. В этом случае необходимая память незаметно выделяется перед исполнением программы.

Массивы

Мы почти готовы создать программу, которая ищет простые числа. Осталось только решить, где они будут храниться. Можно, конечно, использовать для этого стек, но в такой памяти положение чисел относительно вершины постоянно меняется, потому что в стеке временно хранятся регистры и адреса возврата для процедур. Найти простые числа в стеке будет непросто.

Будем поэтому хранить найденные числа в обычной памяти — одно за другим. Если каждому числу выделить участок памяти одного размера (например, два байта), то получить адрес любого из них будет крайне просто, если, конечно, знать адрес начала области памяти и номер. Номер числа мы будем задавать сами, адрес же начала однозначно определяет метка.

Программа, показанная в листинге 4.5, сохраняет 20 чисел (от 1 до 20) в области памяти, начало которой помечено символами simple.

Листинг 4.5. Сохранение чисел от 1 до 20 в памяти

```
.386
.model  flat, stdcall
option  casemap:none
include    \myasm\include\kernel32.inc
includelib \myasm\lib\kernel32.lib
BSIZE equ 20
.data?
simple  dw BSIZE dup(?)
.code
start:
mov ecx,BSIZE
mov bx, 1
mov edi, 0
nxt:
        mov simple[edi],bx
        inc bx
        add edi, 2;переход к следующему числу
```

продолжение ⇨

Листинг 4.5 *(продолжение)*

```
loop nxt
invoke  ExitProcess, 0
end start
```

Директива simple dw BSIZE dup(?) выделяет область памяти, часто называемую *массивом*, для BSIZE идущих подряд 2-байтовых слов. Переписать число из регистра bx в нулевое[1] слово этой последовательности можно инструкцией mov simple,bx. Для доступа к первому, второму и т. д. слову в ассемблере есть специальный способ адресации, примененный в программе из листинга 4.5. Предположим, что регистр edi хранит адрес слова, вычисленный относительно начала массива. Тогда само слово будет выглядеть как simple[edi]. Пусть, например, edi равен 2. Тогда инструкция mov simple[edi],bx записывает содержимое bx в первое слово массива. Если edi равен 0, запись идет в нулевое слово, если равен 4 — во второе и т. д.

В программе из листинга 4.5 мы записываем числа от 1 до 20 в последовательные ячейки массива. Чтобы перейти к следующей ячейке, edi увеличивается на 2, ведь в нашем массиве хранятся 2-байтовые слова.

Но чтобы получить доступ, скажем, к пятой ячейке, совсем не обязательно проходить через предыдущие четыре. Например, запись числа 19 в пятый элемент нашего массива будет выглядеть так:

```
mov edi, 5    ;номер элемента массива
add edi, edi  ;умножаем на 2
mov simple[edi], 19
```

В этом фрагменте edi удваивается[2], потому что наш массив хранит 2-байтовые слова. Если бы там были двойные слова, пришлось бы научиться умножать edi на 4.

[1] Нумерацию в массиве будем вести с нуля. Если, скажем, в массиве 10 чисел, то их номера будут 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

[2] Инструкция add edi, edi велит процессору сложить edi + edi и результат снова послать в edi. Итог этой операции — edi, умноженный на 2.

Кроме edi для доступа к элементам массива можно использовать те же регистры, что и при косвенной адресации (см. раздел «Косвенная адресация» главы 3), то есть eax, ebx, ecx, edx, ebp, esi. Вообще, доступ к элементам массива можно рассматривать как расширенную косвенную адресацию, ведь инструкцию mov simple[edi], bx можно, оказывается, переписать как mov [simple+edi], bx. А это, по сути, косвенная адресация, где адрес в квадратных скобках, равен сумме адреса, связанного с меткой, и относительного адреса (относительно начала массива), хранящегося в регистре edi.

Если верна инструкция mov [simple+edi], 19, то должен быть смысл и в инструкции mov [simple], 19. Логично предположить, что так записывается число в нулевой элемент массива. Но мы уже знаем, что это можно проделать инструкцией mov simple, 19. Значит, метка в квадратных скобках так же хороша, как и метка без них. В любом случае ассемблер будет считать, что это адрес компьютерной памяти.

Простые числа

К этому разделу мы готовились на протяжении всей главы. Но вряд ли программа из листинга 4.6, находящая простые числа, покажется вам такой уж простой.

Листинг 4.6. Вычисление простых чисел

```
.386
.model  flat, stdcall
option  casemap:none
include    \myasm\include\kernel32.inc
includelib \myasm\lib\kernel32.lib
SSIZE  equ 1000
.data?
smpl     dd SSIZE dup(?)
.code
start:
mov ebx, 3  ;первое проверяемое число = 3
```

продолжение ⇨

Листинг 4.6 *(продолжение)*

```
mov edi, 0  ;нулевой элемент массива
mov ebp, 0  ;счетчик простых чисел = 0
nxtdig:
mov edx, 0  ;готовим число edx:eax
mov eax, ebx;к проверке
mov ecx, ebx;число проверок меньше
sub ecx, 2  ;проверяемого числа на 2
mov esi, 2  ;первый делитель = 2
nxtpr:
div esi     ;делим число edx:eax на esi
cmp edx, 0  ;остаток = 0 ?
jz  skip    ;да - идем к след. проверке
mov edx, 0  ;нет -
mov eax, ebx;восстанавливаем edx:eax
inc esi     ;и делим на следующее число
loop nxtpr  ;есть на что делить - продолжим
mov smpl[edi], ebx ; нет - число простое
inc ebp     ;увел. счетчик прост. чисел
cmp ebp, SSIZE;все простые числа найдены?
jz  done    ;да - уходим
add edi,4   ;нет - след. эл-т массива
skip:
inc  ebx    ;проверяем
jmp nxtdig  ;след. число
done:
invoke  ExitProcess, 0
end start
```

Прежде всего, впечатляет ее размер — и это при том, что программа только вычисляет простые числа, но уже не в силах вывести их на экран.

К сожалению, ассемблерные программы очень длинны, во-первых, потому, что любое осмысленное действие требует нескольких инструкций процессора, а во-вторых, из-за того, что в ассемблере мало возможностей уместить в одной строке несколько команд.

Одна строка программы обычно состоит из команды (такой как mov), операндов (это регистры и участки памяти) и комментариев, находящихся правее от точки с запятой.

Комментарии играют важную роль в ассемблере, потому что без них программа непонятна даже ее создателю. Обычно комментируют каждую строчку. Но если ничего путного в голову не приходит, лучше промолчать. Ведь комментарии типа

```
cmp edx, 0  ;равен edx нулю?
```

по меньшей мере, бесполезны, потому что не сообщают программисту ничего нового. Нужно комментировать задачу, а не инструкции ассемблера. Поэтому комментарий

```
cmp edx, 0  ;остаток равен нулю?
```

гораздо лучше.

Для изучения такой программы, как в листинге 4.6, можно использовать несколько стратегий. При этом нужно понимать, что любая стратегия, даже самая глупая, ведет к успеху, если обладаешь бесстрашием и упорством.

Первая разумная стратегия состоит в том, чтобы, пользуясь комментариями, пытаться прокрутить программу «всухую», мысленно выполняя каждую инструкцию. Очень помогает понять программу отладчик, потому что он страхует от ошибок понимания и на каждом шаге помогает увидеть результат ее работы. После длительного кружения в циклах, совершения переходов и наблюдения за меняющимися участками памяти, в мозгу программиста начинает брезжить свет. У него появляется смутная догадка — что же все это значит. Если догадка верна, программист получает «печку», от которой затем и «пляшет». Зная, что делает та или иная инструкция, можно проследить, откуда ей передано управление и куда отправляется процессор после ее выполнения. Постепенно в голове всплывает вся логика работы программы.

Вторая стратегия понимания — частный случай первой. Особенно хороша она, если в программе понятные комментарии. Суть стратегии в том, чтобы сразу выделить в программе центральное место и уже «плясать» от него. В нашей программе таким местом можно считать запись очередного простого числа в память:

```
mov smpl[edi], ebx ;нет - число простое
```

Увидев эту инструкцию, следует понять, как пришла к ней программа. Глядя на исходный текст, убеждаемся, что к этой инструкции можно прийти, только пройдя цикл

```
nxtpr:
div esi       ;делим число edx:eax на esi
cmp edx, 0    ;остаток = 0?
jz  skip      ;да - идем к след. проверке
mov edx, 0    ;нет -
mov eax, ebx  ;восстанавливаем edx:eax
inc esi       ;и делим на следующее число
loop nxtpr    ;есть на что делить -
              ;продолжим
```

в котором, очевидно, и проверяется, простое число или нет. Проверка состоит в том, что число, хранящееся в паре регистров edx:eax, делится на esi. Если остаток, оказавшийся в edx, равен нулю, число делится нацело, следовательно, оно непростое и дальнейшие проверки бессмысленны. В этом случае процессор покидает цикл, переходя к метке skip. Если же остаток не равен нулю, необходима следующая проверка, но регистры edx:eax «испорчены» делением, там нашего числа уже нет. Поэтому нужно снова переписать туда число, сохраняемое в ebx, что и делают инструкции

```
mov edx, 0
mov eax, ebx.
```

затем увеличить делитель командой inc esi и перейти к следующей проверке (loop nxtpr).

Поняв, как работает основной цикл программы, легко догадаться о назначении инструкций, его окружающих. Конечно же, они обслуживают этот внутренний цикл, готовят регистры к тому, чтобы он работал правильно, устанавливают начальные значения и, конечно, следят, нашла ли программа SSIZE все простые числа (инструкция cmp ebp, SSIZE). Если да, пора заканчивать работу (jz done), если же нет — нужно продолжить. А для этого необходимо перейти к следующему элементу массива (add di,4) и получить новое число для проверки (inc ebx).

Задача 4.1. Дополните программу из листинга 4.6 инструкциями вывода простых чисел на экран.

Как пишутся программы

> Татарский вздрогнул. Мысли, туманившие его голову, разлетелись в мгновение ока, и наступила устрашающая ясность.
>
> *В. Пелевин. «Generation П»*

Самое «страшное» для программиста — чистый экран монитора. По-своему, он совершенен, и первые инструкции обязательно нарушат его гармонию. Задача программиста — создать текст, который смотрелся бы не хуже, чем пустой экран.

Начинать следует с общего представления о программе. Нужно прикинуть, сколько в ней будет частей, а потом задуматься о каждой отдельной части. Например, программа вычисления простых чисел может состоять из самого вычисления и вывода простых чисел на экран.

Вычисление, в свою очередь, должно состоять из двух циклов: внешний будет перебирать проверяемые

числа, а внутренний — осуществлять саму проверку. Вырисовываются такие контуры программы:

```
nxtdig:
nxtpr:
...
loop nxtpr
...
loop nxtdig
```

Во внутреннем цикле должны, очевидно, проходить деление и проверка, равен ли нулю остаток. Если не равен, проверка продолжается, если равен — число не простое и нужно выйти из внутреннего цикла во внешний. С учетом сказанного, набросок внутреннего цикла станет таким:

```
nxtpr:
div esi
cmp edx,0
jnz skip
...
inc esi
loop nxtpr
```

Для хранения делителя нам пришлось выделить регистр esi, чьи начальные значения должны устанавливаться вне цикла. Присваивание начальных значений внутри цикла — типичная ошибка программистов, и нужно следить, чтобы туда не попало ничего лишнего.

Во внутреннем цикле появились первые инструкции, и это сразу порождает новые проблемы, что, безусловно, хорошо, ведь теперь нам некогда пугаться — нужно эти проблемы решать.

Первым делом научимся восстанавливать регистры edx, eax, которые «портятся» при каждом делении (в edx посылается остаток, а в eax — частное). Придется выделить для хранения числа один из незанятых пока регистров, например, ebx. С учетом сказанного внутренний цикл станет таким:


```
nxtpr:
div esi
cmp edx, 0
jnz skip
mov edx, 0
mov eax, ebx
inc esi
loop nxtpr
```

Теперь пора подумать о внешнем цикле. Прежде всего, нужно задать правильное значение регистра ecx, чтобы внутренний цикл крутился нужное число раз. Это число, очевидно, на 2 меньше проверяемого, потому что деление на единицу и на само число не имеет смысла. Зная, что проверяемое число находится в регистре ebx, вычислим ecx:

```
mov ecx, ebx
sub ecx, 2
nxtpr:
div esi
cmp edx, 0
jnz skip
mov edx, 0
mov eax, ebx
inc esi
loop nxtpr
```

Заодно с ecx можно задать и начальное значение проверяемого числа (оно хранится в ebx и постоянно переписывается в регистры edx:eax). Начнем проверку с числа 3, потому что двойка заведомо не «проста»:

```
mov ebx, 3
mov edx, 0
mov eax, ebx
mov ecx, ebx
sub ecx, 2
nxtpr:
...
jnz skip
...
loop nxtpr
```

Настало время подумать о метке skip, куда программа отправится в случае деления нацело. Эта метка не должна располагаться сразу после выхода из внутреннего цикла, потому что нормальное его завершение означает, что число простое, то есть сразу после цикла должны быть инструкции, сохраняющие его в памяти компьютера (на все простые числа регистров, конечно, не хватит).

А за меткой skip должны находиться инструкции, которые готовят новое число для проверки, а также выясняют, все ли простые числа найдены. В зависимости от этого программа переходит к новой проверке или завершает работу. И тут нас посещает идея: не будем использовать инструкцию loop для организации внешнего цикла, это хлопотно и потребует сохранять в стеке регистр ecx, «портящийся» во внутреннем цикле. Вместо этого зададим максимальное количество SSIZE и счетчик уже найденных простых чисел (пусть это будет еще не занятый регистр ebp). Тогда «костяк» программы станет таким:

```
nxtdig:
...
nxtpr:
...
jz  skip
...
loop nxtpr
...
cmp ebp, SSIZE
jz  done
skip:
inc  ebx          ;проверяем
jmp nxtdig        ;след. число
done:
```

Набросок программы почти готов. Нам осталось задать массив, где простые числа будут храниться. Как ни странно, куча времени уходит на выбор его имени. Но это время тратится не напрасно, потому что верное имя делает программу более понятной. К тому же, раздумья помогают запомнить имя и привыкнуть к нему.

Я выбрал имя smpl — сухой остаток от слова SIMPLE (простой). Будь эта книжка пошире, я бы, пожалуй, не стал экономить буквы, но в строке программы нужно еще уместить комментарии, поэтому приходится жертвовать наглядностью.

Зная имя массива, можно написать инструкции, сохраняющие в нем найденное простое число. Выделим специальный регистр edi для относительных координат следующего сохраняемого числа. Его начальное значение равно нулю и должно увеличиваться на 4 (в массиве хранятся 4-байтовые числа) при каждой записи простого числа.

С учетом сказанного инструкции записи в массив будут такими:

```
...
mov smpl[edi], ebx
add edi, 4
inc ebp
cmp ebp, SSIZE
jz  done
skip:
...
```

Теперь можно написать всю программу целиком: включить нужные файлы директивами include и includelib, задать начальные значения переменных, вызвать завершающую процедуру ExitProcess и т. д. В результате получится примерно то же, что и в листинге 4.6. Но программирование не терпит «примерности». Любая ошибка может оказаться фатальной. Вот почему программу нужно еще раз просмотреть и только после этого «отдать на съедение» компилятору.

Здесь нас, как правило, подстерегают неожиданности. Компилятор находит множество неправильных или отсутствующих имен, неверных инструкций и т. д. После исправления ошибок он, наконец, благоволит программе и создает исполнимый файл с расширением .exe. Но это совсем не значит, что программа работает правильно. Получив неверный результат, мы снова изучаем ее текст и устраняем замеченные ошибки. Огромную помощь в этом оказывает отладчик. Частое чтение исходного кода притупляет бдительность. Многое мы перестаем замечать. Но отладчик рассеивает иллюзии, и наступает «устрашающая ясность».

И вот после очередной переделки приходит счастливое мгновение: программа выдает ожидаемые числа. Но совсем не потому, что она безошибочна. Если ошибки есть в таких программах, как MS Word или Windows, то почему их не должно быть у нас? Говорят, что в любой программе есть хотя бы одна ошибка. Некоторые ошибки очень коварны и обнаруживают себя, лишь когда программа устарела и должна быть заменена новой, содержащей кучу других ошибок.

Задача 4.2. Найдите хотя бы одну ошибку в программе, вычисляющей простые числа (см. листинг 4.6).

ГЛАВА 5 Шире круг



Логические инструкции

> Бросая в воду камешки, смотри на круги, ими образуемые, иначе такое бросание будет пустою забавою.
>
> *Козьма Прутков. Мысли и афоризмы*

В предыдущей главе нам удалось написать первую настоящую программу. Эта первая программа очерчивает узкий круг основных знаний и навыков, расширяя который можно выучить весь язык. В этой главе мы будем приобретать новые знания, вспоминая о найденных в предыдущей главе простых числах.

И первым делом подумаем о частном случае проверки на «простоту» — исследовании на четность. Выяснить, делится ли число на 2, можно экспериментально — поделив его на 2 инструкцией div и сравнив остаток с нулем. Но можно поступить иначе, вспомнив о представлении числа в виде суммы степеней двойки (двоичном коде). Ясно, что четность и нечетность определяется самым младшим битом, ведь все остальные степени двойки заведомо делятся на 2. Например, число $5 = 2^2 + 1 = 1{*}2^2 + 0{*}2^1 + 1{*}2^0$ нечетно, потому что младший разряд его двоичного представления равен единице, а число 4 четно, потому что этот разряд равен у него нулю.

Но как добраться до отдельного бита, если процессор знает только адреса байтов? Здесь помогут специальные логические операции, в которых участвует каждый бит двух операндов. Для проверки на четность лучше всего подойдет логическое И. Результат этой операции равен единице, когда оба бита равны единице. Во всех других случаях получается ноль. На рис. 5.1 показан результат побитового логического И двух байтов — один хранится в регистре al, второй — в ah.

```
                ah  01010011
and ah, al      al  11000001
                ah  01000001
```

Рис. 5.1. Логическое побитовое И (and)

Инструкция процессора and, выполняющая логическое И, проделывает это самое И над каждой парой бит. Оба младших бита в регистрах ah и al (рис. 5.1) равны единице, следовательно, равен единице и младший бит результата[1]. Следующий по важности бит регистра ah равен 1, а соответствующий бит в al – нулю. Следовательно, бит результата тоже будет нулевым.

Очевидно, логическое И таково, что биты результата окажутся равными нулю, если равны нулю биты одного из операндов. Это свойство логического И легко использовать для выделения младшего бита числа. Создадим специальное число-маску, у которого отличен от нуля только самый младший бит. Тогда результат будет равен либо единице (когда младший бит числа равен единице), либо нулю. При единичном результате число нечетно, при нулевом, понятное дело, четно. Программа, показанная в листинге 5.1, выводит на экран слово Четно или слово Нечетно в зависимости от того, какое число хранится в регистре ah.

Листинг 5.1. Проверка на четность

```
.386
.model  flat, stdcall
option  casemap:none
include    \myasm\include\windows.inc
include    \myasm\include\kernel32.inc
```

продолжение ⇨

[1] Результат логической операции окажется, как обычно, в левом операнде, то есть в регистре ah.

Листинг 5.1 (продолжение)

```
includelib \myasm\lib\kernel32.lib
.data
z        db «Четное».13.10
nz       db «Нечетное».13.10
stdout         dd ?
cWritten       dd ?
.code
start:
invoke  GetStdHandle. STD_OUTPUT_HANDLE
mov stdout. eax
mov ah. 37
and ah. 00000001b    ;выделяем младший бит
cmp ah. 0            ;четно?
jz  evn              ;да - идем к evn
invoke  WriteConsoleA. stdout. ADDR nz. \
            sizeof nz. ADDR cWritten. NULL
jmp  exit
evn:
invoke  WriteConsoleA. stdout. ADDR z.\
             sizeof z. ADDR cWritten. NULL
exit:
invoke  ExitProcess. 0
end start
```

Самая важная инструкция программы and ah.00000001b выделяет младший бит регистра ah с помощью маски 00000001b. Логическое И регистра ah и этой маски даст результат (записанный в ah), у которого все биты, кроме младшего, заведомо равны нулю, а младший бит — такой же, как в регистре ah до операции. Иными словами, единичный результат говорит о нечетности числа, нулевой — о четности.

Операция and ah. 00000001 не очень удобна, потому что «портит» исследуемое число. Чтобы этого не произошло, используется инструкция test, которая, подобно инструкции cmp, искусно «притворяется», что выполняет логическое И. При этом испытуемое число не меняется — меняются только флаги, причем так, как будто операция И на самом деле произошла. С помощью инструкции test фрагмент нашей программы


```
and ah. 00000001b ;выделяем младший бит
cmp ah. 0         ;четно?
```

Перепишется так:

```
test ah. 000001b
```

Задача 5.1. Напишите программу, которая проверяет, делится ли заданное число на 4.

Кроме логического И процессор способен выполнить две другие логические операции: ИЛИ (инструкция or), а также исключающее ИЛИ (инструкция xor).

Операция ИЛИ гораздо менее требовательна, чем И. Ее результат равен единице, когда равен единице хотя бы один бит. На рис. 5.2 показаны операнды, уже знакомые нам по примеру с операцией И (см. рис. 5.1). На этот раз с ними совершается побитовое логическое ИЛИ.

	ah	01010011
or ah, al	al	11000001
	ah	11010011

Рис. 5.2. Логическое побитовое ИЛИ

Операция логического ИЛИ бывает полезна, когда необходимо объединить несколько условий, закодированных отдельными битами. Если младший бит одного регистра, установленный в единицу, показывает, что число делится на два, а следующий по значимости бит другого регистра говорит о том, что число делится на 3, то после операции ИЛИ над обоими регистрами станет ясно, что число делится и на 3 и на 2.

Смысл другой побитовой логической операции xor (исключающего ИЛИ), часто обозначаемой значком A, тоже прост: операция xor выделяет различия в регистрах. Там где биты одинаковы, получается ноль, там где различны — единица. Для побитового исключающего ИЛИ справедливы правила 1 A 1 = 0, 0 A 0 = 0, 1 A 0 = 1, 0 A 1 = 1. На рис. 5.3 показан результат операции поби-

тового исключающего ИЛИ над уже привычными нам операндами.

```
            ah  01010011
xor ah, al  al  11000001
            ah  10010010
```

Рис. 5.3. Исключающее ИЛИ показывает различия в регистрах

Очень часто в программах на ассемблере можно встретить странные инструкции xor с одинаковыми операндами, например xor eax, eax. Легко понять, что таким образом eax просто приравнивается нулю, ведь биты в одном и том же регистре попарно равны, следовательно, результат операции исключающего ИЛИ над каждой парой также будет нулевым. Программисты используют xor отчасти из пижонства, ведь xor eax, eax смотрится гораздо «круче» тривиального mov eax, 0, отчасти из-за того, что инструкция xor eax, eax занимает всего два байта (33C0), а mov eax, 0 — целых 5 (B800000000). Кроме того, исключающее ИЛИ процессор может выполнить быстрее, а скорость и компактность очень важны для программ на ассемблере.

Задача 5.2. Что делают инструкции

```
xor eax,ebx
xor ebx,eax
xor eax,ebx?
```

Подсказка: рассмотрите четыре комбинации: eax = 1, ebx = 0; eax = 1, ebx = 1; eax = 0, ebx = 0; eax - 0, ebx = 1.

Сдвиги

В предыдущем разделе мы научились понимать, четное ли перед нами число, «погасив» все его биты, кроме самого младшего. Оказывается, выделить младший бит можно и с помощью инструкции shr, которая перемещает старшие биты на указанное число позиций вправо. При этом биты «сваливаются» с правого конца сдвигаемого числа, но последний свалившийся бит не пропадает, а сохраняется во флаге переноса C. Пусть, например, регистр ah сдвигается на один шаг вправо. Соответствующая инструкция выглядит в этом случае так:

```
shr ah, 1
```

При этом биты, из которых состоит число, перемещаются следующим образом: нулевой (самый младший) оказывается во флаге переноса, первый переходит на место нулевого, второй — на место первого, ... седьмой на место шестого, а на место самого старшего седьмого бита процессор ставит ноль (рис. 5.4).

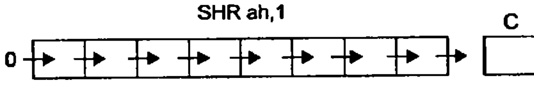


Рис. 5.4. Сдвиг вправо

С помощью сдвига вправо проверка числа, хранящегося в регистре ah, на четность выглядела бы так:

```
shr ah, 1   ;загоняем младший бит в C
jc  odd     ;мл. бит равен 1 - нечетное
```

Здесь использована другая условная инструкция jc odd, отправляющая процессор к метке odd, когда поднят флаг переноса C. В противном случае процессор выполнит инструкцию, следующую за jc odd. Инструкций, подобных jc или jnz, очень много, потому что велико разнообразие событий, происходящих при выполнении программы. Все инструкции условного перехода способны отбросить процессор либо на 127,

либо на 32 767 байт в сторону от себя[1]. Для переходов на большее расстояние нужно использовать комбинацию условной инструкции и безусловного перехода jmp.

Задача 5.3. Подумайте, как совершить условный переход на любое расстояние.

Задача 5.4. Напишите программу, подсчитывающую число единичных бит в регистре.

Но давайте вернемся к инструкциям сдвига. Кроме выталкивания битов во флаг переноса, они, оказывается способны делить и умножать числа на степени двойки. Сдвиг вправо на одну позицию эквивалентен делению на 2, а сдвиг влево — умножению на 2.

Действительно, двоичное число представляется суммой степеней двойки. Причем вес соседних битов отличается вдвое. Возьмем, к примеру, число 7, равное 4 + + 2 + 1 или в двоичном представлении 111. В нем вес старшего бита равен 2^2, то есть 4, вес следующего 2^1, то есть 2. Если семерку сдвинуть на шаг вправо, то четверка превратится в двойку, двойка — в единицу, а единица и вовсе скроется во флаге переноса. В результате получится число 3, и это значит, что сдвиг вправо выполняет деление нацело, остаток от деления вытесняется за пределы числа и сохраняется только при сдвиге на один бит во флаге переноса.

Задача 5.5. Как найти остаток от деления числа на степень двойки?

Если сдвиг вправо делит число, то сдвиг влево, понятное дело, умножает его на степень двойки, равную

[1] Так происходит потому, что длина «прыжка» кодируется в этих инструкциях либо байтом, способным хранить числа от –128 до +127, либо словом, вмещающим числа от –32768 до +32767. Какую инструкцию выбрать — решает ассемблер. Если «прыжок» получается меньше чем на 127 байтов, он выбирает более короткую инструкцию, если нет — более длинную.

числу сдвигов. Сдвиг влево на N позиций умножает число на 2^N. Сдвигом влево ведает в языке ассемблера инструкция shl. Подобно инструкции shr она выдавливает биты во флаг переноса, но только с противоположного конца регистра (участка памяти). Число сдвигов можно задавать не только явно, но и в регистре cl[1]. Инструкции, показанные ниже, сдвигают регистр eax влево на две позиции:

```
mov eax.7
mov cl.2     ;задать число сдвигов
shl eax.cl   ;умножить на 2^cl.
```

После выполнения инструкции shl eax. cl в регистре eax окажется число 28.

Сдвиги влево можно сочетать с командами сложения и тогда станет возможным умножение на почти любое число. Например, умножение eax на 3 выполнят следующие инструкции:

```
mov ebx. eax     ;запоминаем eax
shl eax. 1       ;eax умножается на 2
add eax. ebx     ;eax становится втрое
                 ;больше
```

Задача 5.6. Напишите программу, в которой заданное число умножается на 10 с помощью сдвигов и сложений.

Говоря о сдвигах, мы до сих пор предполагали, что имеем дело с положительными числами. Сдвиг отрицательных чисел удобно изучать на примере 4-битовых регистров, уже знакомых нам по главе 2.

Первым делом посмотрим, как выполняется сдвиг отрицательного числа влево, поскольку для него достаточно уже известной нам инструкции shl, ведь «значимые» биты отрицательного числа расположены в

[1] Это справедливо и для инструкции shr.

правой его части, а левую часть заполняют единицы, и потеря одной из них ни к чему страшному не приведет. Пусть, например, на шаг влево сдвигается число −1. В 4-битовом регистре оно представляется четырьмя единицами 1111. После сдвига влево получится число 1110, что эквивалентно в дополнительном коде числу −2 (проверьте это!).

Чтобы убедиться в том, что сдвиг влево безопасен, попробуем представить его в общем виде. Если k — отрицательное число, то его дополнительный код для 4-битового регистра равен 16 − |k|, где | | — знак модуля, то есть абсолютной величины числа. Когда число со знаком сдвигается влево, его знаковый бит выдавливается во флаг переноса, то есть из числа вычитается 8, а все, что осталось, умножается на 2. В итоге получаем −(16 − |k| − 8) * 2 = −(16 − 2 * |k|). То есть при сдвиге отрицательного числа влево на один шаг оно умножается на два. Поскольку любой сдвиг можно представить последовательностью «единичных» сдвигов, команда shl годится для сдвига отрицательных чисел на любое число позиций.

Правда, нужно следить, чтобы сдвигаемое число уместилось в регистре. Пытаясь сдвинуть число −5, записанное в 4-битовом регистре, получим ерунду, потому что −10 никак не помещается в четырех битах.

Задача 5.7. Какие отрицательные числа можно сдвигать на шаг влево в 8-, 16- и 32-битовых регистрах?

Если сдвиг влево отрицательного числа неотличим от сдвига положительного и выполняется одной и той же инструкцией shl, то команда shr, которая уже применялась для сдвига вправо положительного числа, явно испортит число отрицательное, потому что обратит знаковый бит в ноль. Поэтому для сдвига вправо отрицательных чисел применяется другая инструкция sar. Она работает так же, как и shr, но заполняет опустевшие старшие биты не нулями, а единицами.

Пусть, например, требуется сдвинуть на шаг вправо число −5. Поскольку отрицательные числа в 4-битовых регистрах получаются дополнением до 16 (см. раздел «Знак» главы 2), число −5 кодируется так же, как положительное число 11_{10}: 1011_2. После сдвига на шаг вправо получим 0101, а после записывания единицы на место знакового бита 1101, то есть 8 + 4 + 0 + + 1 = 13 — представление в дополнительном коде числа −3. То есть команда sar округляет отрицательные числа в другую сторону, и мы получаем на единицу больше, чем ожидали. Например, применив инструкцию sar к числу −1, получим снова −1, а не ожидаемый ноль.

Чтобы окончательно убедиться в том, что инструкция sar работает верно, попробуем представить то, что она делает, в общем виде. Пусть k — отрицательное число, записанное в дополнительном коде. Если говорить о 4-битовых регистрах, то это будет дополнение до 16, то есть 16 − |k|. Сдвиг на одну позицию командой sar, примененный к 4-битовому регистру, можно представить как shr(16 - |k|) + 8, где shr — обычный сдвиг на позицию вправо, а добавление восьмерки — это запись единицы в старший разряд 4-битового регистра. Чтобы понять, какое отрицательное число вышло в результате, нужно из шестнадцати вычесть результат работы команды sar:

```
-(16 - shr(16 - |k|)-8)) = -(8 - shr(16 - |k|)).
```

Пусть, например, сдвигается число −7. Тогда |k| = |−7| = 7. 16 − |k| = 9. Обычный сдвиг вправо числа 9 даст 4, то есть -(8 -shr(16 - |k|)) = -4. Итак, сдвиг командой sar на одну позицию числа −7 даст нам −4, записанное в дополнительном коде.

Круженье бит

Для выполнения изделия использовали большое количество бусин. Например, на вышивание одной стороны кошелька среднего размера уходило около 10 000 бисерин. Эта кропотливая работа требовала особого внимания, терпения, а главное, любви.

Татьяна Косоурова.
«Бисер в культуре народов мира»

Программирование на ассемблере, как и вышивание бисером, связано с кропотливой подгонкой множества инструкций друг к другу. Подобно кошельку среднего размера, программа на ассемблере может содержать тысячи «инструкций-бисерин».

В этой книге нам, конечно, не придется работать с длинными программами. Но чтобы получить представление об ассемблере, достаточно гораздо более коротких текстов.

Подумаем, например, как хранить и выводить на экран текущую дату — число, месяц и год. Эту задачу можно решить «в лоб», выделив для хранения дня, месяца и года три идущих подряд переменных. Но жалко тратить целый байт (не говоря о слове) на хранение таких небольших чисел. Попробуем поэтому понять, сколько всего нужно бит для хранения даты.

Номер дня (число) не превышает 31 и поместится в пяти битах[1]. Месяц уместится в четырех битах, а год, если брать только две последние цифры (от 0 до 99) — в семи. Выходит, для хранения даты достаточно 5 + 4 + + 7 = 16 бит или машинного слова (рис. 5.5).

[1] Вспомним, что 4 бита могут быть в 16, то есть в 2^4 разных состояниях. Значит, 5 битов способны хранить $2^5 = 32$ различных числа.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D	D	D	D	D	M	M	M	M	Y	Y	Y	Y	Y	Y	Y

Рис. 5.5. Дата умещается в 16 битах: D — день, M — месяц, Y — год

Остается только понять, как на практике втиснуть дату в шестнадцать бит и как менять, скажем, день, не касаясь месяца и года.

Самый очевидный путь — сдвигать нужное число во вспомогательном регистре и затем «наклеивать его» в нужное место с помощью логической операции ИЛИ (or). Фрагмент программы, задающей дату 30 июля 2003 (03) года, может выглядеть так:

```
; 30 июля 2003 года
xor ax, ax   ;ax = 0
xor bx, bx   ;bx = 0
mov bx, 3    ;записываем год
or  ax, bx   ;в ax
mov bx, 7    ;устанавливаем
shl bx, 7    ;месяц
or  ax, bx   ;записываем месяц
mov bx, 30   ;устанавливаем
shl bx, 11   ;день
or  ax, bx   ;записываем день
```

В нем используется вспомогательный регистр bx, где формируется нужная составляющая даты. Чтобы, например, задать 2003 год, нужно записать число 3 в регистр bx и затем «наклеить» его на регистр ax логическим ИЛИ (or). Год автоматически занимает нужные биты в ax, потому что кодируется семью младшими битами. С месяцем так не получится. После его задания в регистре bx (это 7 — номер июля) биты нужно сдвинуть влево на 7, чтобы они заняли позиции 7–10 (рис. 5.5), и лишь затем «наклеить» их в регистр ax. Наконец, день месяца (30) так же задается в регистре bx, но биты нуж-

но уже сдвинуть на 11 позиций, чтобы они заняли биты 11–15 (рис. 5.5).

После того как дата сформирована, нужно подумать о ее модификации. Эта задача уже сложнее, потому что нужно в общем случае менять биты, стоящие посередине или у левого края регистра. Здесь помогут специальные инструкции циклического сдвига rol (влево) и ror (вправо), до сих пор нам не встречавшиеся.

В отличие от обычных сдвигов shl и shr, циклические сдвиги не сталкивают биты с края, а сохраняют их в противоположном конце регистра. Если, например, обычный сдвиг влево shl выталкивает старший бит из регистра во флаг переноса, то циклический сдвиг на шаг приводит к тому, что четырнадцатый бит занимает позицию пятнадцатого, тринадцатый бит встает на место четырнадцатого, ...нулевой на место первого. А место нулевого бита занимает самый старший, пятнадцатый. Результат циклического сдвига даты, хранящейся в регистре ax, на 5 позиций влево показан на рис. 5.6.

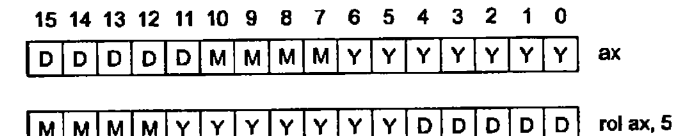


Рис. 5.6. Циклический сдвиг даты на 5 позиций влево

После такого сдвига изменить дату гораздо легче: нужно обнулить в циклически сдвинутом регистре первые пять битов, записать новую дату во вспомогательный регистр, «наклеить» ее операцией ИЛИ (or) и, наконец, вернуть дату на прежнее место циклическим сдвигом на то же число позиций, но на этот раз вправо:


```
rol ax, 5       ;перемещаем дату в начало
and ax, 0ffe0h  ;обнуляем дату
mov bx, 31      ;31 число во вспом. регистре
or  ax, bx      ;меняем дату
ror ax, 5       ;дату – на прежнее место
```

Наверное, непонятнее всего в этом отрывке инструкция and ax, 0ffe0h, обнуляющая дату. Нули на месте младших пяти битов нужны, чтобы биты прежней даты не перемешались с битами новой. Если же все биты предварительно обнулить, то операция ИЛИ (or) наклеит новые биты на место старых и путаницы не произойдет. Убедимся теперь, что инструкция and ax, 0ffe0h действительно обращает в ноль первые 5 бит регистра ax. В самом деле, число ffe0 равно 1111 1111 1110 0000 в двоичном представлении, а операция И (and) оставит неизменными те биты регистра ax, которым соответствуют единичные биты маски, и обнулит те биты, которые в маске равны нулю.

Теперь мы в состоянии понять программу, которая записывает дату (30 июля 2003 года) в 2-байтовый регистр, меняет число с 30 на 31 и затем показывает новую дату на экране (листинг 5.2).

Листинг 5.2. Запись и модификация даты

```
.386
.model  flat, stdcall
option  casemap:none
include    \myasm\include\windows.inc
include    \myasm\include\user32.inc
include    \myasm\include\kernel32.inc
includelib \myasm\lib\kernel32.lib
includelib \myasm\lib\user32.lib
DateDisp proto :WORD
BSIZE equ 15
.data
ifmt            db «%d»,0
buf             db BSIZE dup(0)
stdout          dd ?
```

продолжение ⇩

Листинг 5.2 *(продолжение)*

```
cWritten        dd ?
month db «янв фев мар апр май июн «
        db «июл авг сен окт ноя дек»
.code
start:
invoke  GetStdHandle. STD_OUTPUT_HANDLE
mov stdout. eax
mov  ax. 0f383h      ;30 июля 2003 года
invoke DateDisp. ax ;показать дату
invoke  ExitProcess. 0
DateDisp proc Date:WORD
xor edi.edi          ;очищаем регистр
mov di. Date
rol di. 5            ;число - в пять младших
                     ;бит
and di. 1fh          ;гасим лишние биты
invoke  wsprintf. ADDR buf. ADDR ifmt. edi
invoke  WriteConsole. stdout. ADDR buf. 3.\
       ADDR cWritten. NULL
mov di. Date          ;восстанавливаем дату
shr di. 7             ;месяц - в семь
                      ;младших бит
and di. 0fh           ;выделяем месяц
dec di                ;нумерация с нуля
shl di. 2             ;умножим на 4
mov esi. offset month:отн. адрес названия
add esi. edi          ;адрес названия
invoke  WriteConsole. stdout. esi. 4. \
       ADDR cWritten. NULL
mov  di. Date         ;восстанавливаем дату
and  di. 7fh          ;выделяем год
invoke  wsprintf. ADDR buf. ADDR ifmt. edi
invoke  WriteConsole. stdout. ADDR buf. 3.\
       ADDR cWritten. NULL
ret
DateDisp endp
end start
```

Для экономии места я не стал формировать дату с помощью сдвигов и наложений масок, а просто записал в регистр ax число 0f383h (проверьте, действительно ли

оно соответствует 30 июля 2003 года). Прежде чем переходить к процедуре DateDisp, показывающей дату на экране, нужно решить, как удобнее всего показывать месяц. В программе из листинга 5.2 все названия месяцев занимают одно и то же число байтов — четыре, что сильно упрощает вычисление адреса нужных символов, хранящихся в массиве month. Из-за недостатка места пришлось записать этот массив в две строки, каждая из которых начинается директивой db. Как мы уже знаем (см., например, раздел «Не могу молчать» главы 3), ассемблер не считает двойные кавычки символами, кавычки лишь подсказывают ему, где начинаются и где кончаются «настоящие» символы. Обратите внимание на пробел после названия июня — июн «. Его обязательно нужно оставить, иначе нарушится порядок символов и программа перепутает названия месяцев.

Сама процедура DateDisp теперь наверняка покажется нам простой. В ней с помощью сдвигов и битовых масок выделяются и показываются на экране разные составляющие даты. Чуть сложнее, чем день и год, показывается месяц. Чтобы вычислить адрес нужной последовательности символов, необходимо начать нумерацию месяцев с нуля, а не с единицы, как у нас. Поэтому после выделения месяца инструкциями

```
mov di. Date     ;восстанавливаем дату
shr di. 7        ;месяц - в семь младших бит
and di. 0fh      ;выделяем месяц
```

его номер уменьшается на единицу, затем вычисляется адрес первого символа массива month (offset month) и к нему прибавляется номер месяца, умноженный на 4 (потому что каждое название месяца занимает с учетом пробела 4 байта). Зная адрес и число символов, можно запускать процедуру WriteConsole, которая и показывает нужный месяц (в нашем случае июл) на экране.

Сложение и вычитание

В прошлой главе мы не рисковали проверять на «простоту» числа, большие, чем в состоянии хранить 4-байтовый регистр eax, потому что не знали, как разместить число в двух регистрах edx:eax сразу. Оказывается, сделать это можно, прибавляя к регистру eax по единице и следя, не произошел ли перенос из старшего бита eax. Если такое случилось, нужно прибавить единичку к регистру edx и поступать так всякий раз, когда эти переносы возникают.

Следить за переносом способна специальная команда adc, которая прибавляет число, как обычная команда add, а затем добавляет к сумме содержимое флага переноса. С помощью команды adc формирование числа для проверки на «простоту» будет выглядеть так:

```
add eax, 1   ;увеличим
adc edx, 0   ;число
```

Первая инструкция увеличивает на единицу eax, а вторая прибавляет к edx флаг C. Вместо add eax, 1 нельзя использовать инструкцию inc eax, потому что она не влияет на флаг переноса.

Естественно, комбинацию add, adc можно использовать не только для прибавления к «длинному» числу единицы, но и для сложения очень больших чисел, не умещающихся ни в двух, ни в четырех байтах. Пусть, например, одно число хранится в паре регистров edx:eax, а второе в паре ecx:ebx. Для сложения таких чисел понадобятся инструкции:

```
add eax, ebx ;eax <- eax + ebx
adc edx, ecx ;edx <- edx + ecx + C
```

Первая инструкция выполняет обычное сложение, а вторая складывает регистры edx и ecx и прибавляет к их сумме флаг переноса.

Задача 5.8. Подумайте, как можно складывать «длинные» числа, хранящиеся не в регистрах, а в массивах, расположенных в памяти компьютера.

«Длинные» числа можно не только складывать, но и вычитать с помощью команд sub, sbb. Понять их работу невозможно без умения различать сложение и вычитание.

До сих пор мы думали, что вычитание эквивалентно прибавлению числа, записанного в дополнительном коде (см. раздел «Знак» главы 2). Пусть, например, два числа 1 и 3 хранятся в 8-битовых регистрах. Тогда (думали мы) для вычитания из единицы трех достаточно сложить единицу (00000001) и 3 в дополнительном коде (11111101). Действительно, сложив эти числа, получим 11111110, то есть −2 в дополнительном коде (убедитесь в этом сами).

Казалось бы, все верно. Но кроме числа нужно еще правильно установить флаги, а при сложении единицы и дополнительного кода для −3 не возникает ни переполнения, ни переноса из старшего разряда.

Между тем, вычитание из меньшего числа большего, как в нашем примере, должно сопровождаться *заемом* из старшего разряда. Чтобы понять, что это такое, попробуем вычесть из единицы три «в лоб», не прибегая к двоичному дополнительному коду, а руководствуясь обычными правилами вычитания «столбиком»[1] (рис. 5.7)

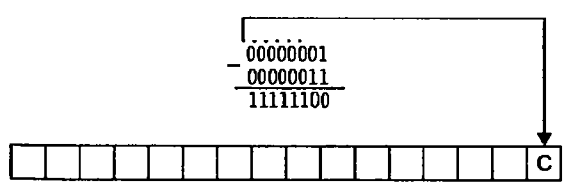


Рис. 5.7. Вычитание «столбиком» двоичных чисел

[1] Точками на рисунке помечены заемы из старших разрядов.

Вычитая самые младшие разряды, получим 0. Переходя на шаг влево, столкнемся с необходимостью вычитать из нуля единицу. Это невозможно, поэтому, как и при вычитании десятичных чисел, займем единицу старшего разряда. Она имеет вдвое больший вес, поэтому вычитание из единицы старшего разряда единицы младшего даст единицу[1]. Далее переходим к вычитанию из нуля, из которого уже занята единица, «обычного нуля». Опять занимаем единицу старшего разряда и опять получаем единицу младшего. Продолжая в том же духе, получим ряд единиц, но последняя единица получается, если занять единицу несуществующего, девятого разряда. Вот это и есть заем, о котором процессор должен сообщать поднятием какого-то флага. В процессорах Intel для этого выбран флаг переноса.

Для большей ясности вернемся к нашему примеру. Если складывать командой add два числа 00000001 (единицу) и 11111101 (–3 в дополнительном коде), получится 11111110 (–2 в дополнительном коде). При этом флаг C окажется опущенным (равным нулю). Если же для вычитания из единицы тройки использовать команду sub:

```
mov al, 1
mov ah, 3
sub al, ah,
```

то результат будет тем же (в регистре al окажется число 11111110), но флаг C будет поднят, что скажет нам о заеме из старшего разряда.

Задача 5.9. На примере 4-битовых регистров докажите, что вычитание sub можно реализовать как сложение с использованием дополнительного кода, но флаг C, устанавливаемый командой add, необходимо инвертировать. Когда есть перенос при сложении, нет заема при вычитании, и наоборот, отсутствие переноса при сложении означает заем при вычитании.

[1] Занятая единица имеет вес 4, а единица текущего разряда имеет вес 2, вычитая, получаем 2, то есть как раз единицу текущего разряда.

Теперь мы, наконец, можем вернуться к вычитанию «длинных» чисел с помощью команд sub и sbb. Предположим, что в регистрах edx:eax записан ноль. Тогда вычесть из такого длинного числа единицу можно командами

```
sub eax, 1
sbb edx, 0
```

Первая команда приводит к тому, что в eax все биты устанавливаются в единицу и возникает заем из старшего разряда (поднимается флаг переноса C). Вторая команда вычитает флаг переноса, равный в нашем случае единице, из числа, записанного в регистре edx. Результат понятен: в edx и eax все биты обратятся в единицу, то есть в паре регистров оказывается –1 в дополнительном коде.

Последовательность команд sub, sbb, sbb... можно применить к вычитанию любых длинных чисел. Пусть, например, первое число находится в паре регистров edx:eax, а второе — в паре ecx:ebx. Тогда фрагмент программы вычитания из первого числа второго будет выглядеть так:

```
sub eax, ebx ;вычитаем «младшие» биты
sbb edx, ecx ;edx <- edx - ecx - C
```

Первая инструкция sub вычитает из eax содержимое ebx и записывает результат в eax, она занимается «младшими» битами. Вторая инструкция посылает в edx результат операции edx - ecx - C.

Умножение и снова деление

> В контрольном эксперименте баран выбрал правую кормушку. Собственно, задача сводилась к вопросу: почему? Два года машина думала. Потом начала строить модели.
>
> *Аркадий и Борис Стругацкие. Полдень. XXII век*

Мы уже говорили о том, что умножение почти на любое число можно представить комбинацией сдвигов и сложений. Но было бы странно делить числа инструкцией div, а умножать, изобретая каждый раз хитрые комбинации инструкций add и shl.

Поэтому в процессоре предусмотрена инструкция универсального умножения mul Как и div, инструкция mul <операнд> едина в трех лицах: операнд, хранящийся в байте, умножается на al, результат же оказывается в регистре ax. Если операнд — слово, процессор умножает его на ax, результат же оказывается в eax. Наконец, операнд, хранимый в двойном слове, умножается на eax, а результат оказывается в паре регистров edx:eax.

Как обычно, новое знание порождает новую печаль: результат умножения, хранящийся в двух регистрах и занимающий 64 бита, необходимо выводить на экран, а для этого процедура wsprintf не подойдет, потому что она оперирует только 32-битовыми значениями.

Значит, нужно самим научиться превращать «длинные» числа в символы, для вывода которых на экран годится процедура WriteConsoleA. Программа, выводящая на экран «длинные» числа, показана в листинге 5.3.

Листинг 5.3. Вывод на экран «длинных» чисел

```
.386
.model  flat, stdcall
option  casemap:none
include    \myasm\include\windows.inc
```



```
include    \myasm\include\kernel32.inc
includelib \myasm\lib\user32.lib
includelib \myasm\lib\kernel32.lib
BSIZE     equ 20
.data
digit     db BSIZE dup (?)
cWritten  dd ?
stdout    dd ?
.code
start:
mov  esi, BSIZE
mov  eax, 123456789
mov  ebx, 1315678
mul  ebx
mov  ebx, edx
mov  ecx,10           ;делим на 10
nxt:
dec  esi        ;позиция след. символа
xchg eax, ebx   ;делим
sub  edx, edx   ;старшую
div  ecx        ;половину
xchg eax, ebx   ;сохраняем частное и делим
div  ecx        ;остаток + младшую половину
add  dl, 48           ;превращаем в символ
mov  digit[esi], dl   ;сохраняем символ
mov  edx, ebx
or   edx, eax         ;оба частных = 0?
jnz  nxt              ;нет - продолжим
invoke  GetStdHandle, STD_OUTPUT_HANDLE
mov  stdout, eax
mov  eax, offset digit;начало массива
add  eax, esi         ;адрес первого
                      ;символа
mov  edx, BSIZE
sub  edx, esi         ;число символов
invoke  WriteConsoleA, stdout, eax, edx, \
        ADDR cWritten, NULL
invoke  ExitProcess, 0
end start
```

Хоть и не самая длинная, эта программа несравнимо сложнее всего того, что нам до сих пор встречалось. И если хочется понять, за что одни так любят ассемб-

лер, а другие — люто ненавидят, то лучшего примера не найти.

Программисту[1] понадобилось всего несколько инструкций процессора, чтобы реализовать довольно сложный алгоритм. В этом — большая красота ассемблера и в этом же — большое неудобство, ведь понять спрессованную в несколько строчек мысль не так то просто.

Поэтому имеет смысл построить модель, как это делала вычислительная машина из романа Стругацких, и по шагам проследить, что же делают инструкции. Наша модель окажется довольно близкой ассемблерному тексту, но в отличие от него будет оперировать не двоичными, а гораздо более привычными десятичными числами.

Но прежде выделим из листинга 5.2 центральную часть, которая и превращает «длинное» число в последовательность символов:

```
nxt:
…
xchg eax, ebx    ;делим
sub  edx, edx    ;старшую
div  ecx         ;половину
xchg eax, ebx    ;сохраняем частное и делим
div  ecx         ;остаток + младшую половину
…                ;сохранить символ
mov  edx, ebx
or   edx, eax    ;оба частных = 0?
jnz nxt          ;нет - продолжим
```

Чтобы не отвлекаться, я поставил многоточия на месте инструкций, сохраняющих символ в массиве digit. А теперь — наша модель. Пусть в длинном слове, хранящемся в паре регистров ebx:eax, записано десятичное число 123456: в регистре ebx — 123, а в регистре eax — 456. Тогда первая инструкция xchg eax, ebx поменяет

[1] В листинге 5.2 использован текст программы, написанной Ричардом Павличеком (Richard Pavlicek pavlicek@gate.net).

местами eax и ebx. Значит, после того как в edx окажется ноль, мы поделим число 123, оказавшееся в eax, на 10. Результат этого деления — число 3 в edx (остаток) и число 12 в eax (частное). Далее (смотрите исходный текст) следует вторая команда xchg, которая опять меняет регистры eax и ebx. И следующим числом, которое будет поделено на 10, будет 3456 (3 в edx и 456 в eax)! После деления в edx окажется остаток 6 — первый десятичный разряд числа, а в регистре eax — частное 345. Чувствуете? Программа «откусила» десятичный разряд (3) от старшей половинки числа и перевела его в младшую. После чего удалила и сохранила младший десятичный разряд (6). Так она будет действовать и дальше: присоединять десятичный разряд слева и удалять справа, пока частные от деления старшей и младшей половин числа на 10 не станут равны нулю. Это условие проверяется инструкцией or edx, eax, результат которой будет равен нулю, лишь когда равны нулю оба операнда edx (частное от деления старшей половины) и eax (частное от деления младшей половины).

Теперь можно перейти от модели к реальности, то есть к делению не десятичных, а двоичных чисел. Очевидно, все будет так же, но поскольку число 10 не соответствует целому числу бит (3 бита вмещают 7_{10}, а 4 — 15_{10}), количество двоичных разрядов, втягиваемых с одного конца и выталкиваемых с другого, будет переменным. Но суть от этого не изменится, и «длинное» число будет успешно протащено через пару регистров edx:eax, после чего программа выведет на экран «длиннющее» число 1624293812379942, равное произведению 1315678 на 123456789 (см. листинг 5.2).

Читая описание сложного алгоритма, вы, наверное, не раз уже подумали: а не лучше ли просто поделить инструкцией div число, хранящееся в паре регистров edx:eax, на 10, сохранить остаток, затем поделить на 10 частное

от предыдущего деления, снова сохранить остаток и поступать так до тех пор, пока частное не станет равно нулю? Все дело в том, что такое деление может закончиться *переполнением*, когда частное не уместится в регистре eax. Вот почему принципиально важно разбить число на две части, каждую из которых можно безопасно делить хоть на 1, не говоря о десяти, и постепенно перетаскивать кусочки одной части в другую.

Задача 5.10. Напишите процедуру деления 64-битового числа, хранящегося в двух регистрах, на произвольное число. Подсказка: вспомните деление «столбиком» десятичных чисел.

После длинного описания очень короткого алгоритма нам осталось сделать два замечания. Первое касается нового оператора offset, который используется в листинге 5.2 для получения адреса начала массива digit:

```
mov eax, offset digit
```

Оператор offset похож на оператор ADDR, но ADDR применяется только при вызове функции директивой invoke, поэтому и приходится использовать offset, чтобы послать адрес массива в eax. В принципе offset можно использовать вместо addr при вызове процедуры директивой invoke, у оператора ADDR только одно преимущество: он позволяет узнать адрес локальных переменных, выделяемых директивой LOCAL (см. раздел «Своеволие ассемблера» главы 3), а offset — нет.

Второе замечание касается учета знака чисел при умножении и делении. До сих пор нам приходилось только делить положительные числа. Полезно знать, что существуют специальные инструкции для умножения и деления чисел со знаком — imul (умножение) и idiv (деление). Их единственное отличие от div и mul в том, что они рассматривают числа с единичным старшим битом как отрицательные, и соответственно меняют результат умножения или деления.

Ввод

До сих пор мы покорно выслушивали все, что желает сообщить программа, но сами не могли вставить и словечка, потому что не знали, как общаться с программой во время ее выполнения. Пора перейти от грубого вмешательства в исходные тексты к более деликатному вводу символов с клавиатуры.

Этим в системе Windows ведает процедура ReadConsole, одновременно похожая и противоположная уже известной нам WriteConsoleA. Программа, показанная в листинге 5.4, вводит с клавиатуры последовательность символов и затем отображает ее на экране.

Листинг 5.4. Ввод с клавиатуры и отображение на экране символов

```
.386
.model  flat, stdcall
option  casemap:none
include    \myasm\include\windows.inc
include    \myasm\include\kernel32.inc
includelib \myasm\lib\user32.lib
includelib \myasm\lib\kernel32.lib
BSIZE      equ 128
.data
buf             db BSIZE dup(?)
stdout          dd ?
stdin           dd ?
cRead           dd ?
cWritten        dd ?
.code
start:
invoke  GetStdHandle, STD_OUTPUT_HANDLE
mov  stdout, eax
invoke  GetStdHandle, STD_INPUT_HANDLE
mov  stdin, eax
NewLine:
invoke  ReadConsole, stdin, ADDR buf,\
        BSIZE, ADDR cRead, NULL
invoke  WriteConsoleA, stdout, ADDR buf,\
```

продолжение ↻

Листинг 5.4 *(продолжение)*

```
            cRead. ADDR cWritten. NULL
    cmp     cRead.2
    jnz     NewLine
    invoke  ExitProcess. 0
    end start
```

Программа из листинга 5.4, в отличие от всех остальных, имеет дело с дескрипторами *двух* стандартных устройств — экрана (stdout) и клавиатуры (stdin). Процедура ReadConsole принимает, по существу, те же параметры, что и WriteConsoleA: дескриптор устройства (stdin), адрес массива байтов, куда попадут введенные символы (ADDR buf), размер массива (BSIZE), адрес двойного слова, куда процедура запишет число прочитанных символов (ADDR cRead), и, наконец, ничего не значащее значение NULL (просто ноль).

Ввод символов завершается клавишей Enter. После ее нажатия ReadConsole заканчивает работу и за дело берется процедура WriteConsoleA, повторяющая введенные символы на экране.

Задача 5.11. Напишите процедуру, переводящую, когда это возможно, последовательность символов в число.

Нужно понимать, что Enter тоже вводит с клавиатуры символы, а не просто управляет вводом. Символов этих два: 13 ($0D_{16}$) и 10 ($0A_{16}$). Первый из них обозначается как CR (*Carriage Return — возврат каретки*), а второй — как LF (*Line Feed — перевод строки*). Легко догадаться, что первый возвращает курсор к левому краю экрана, а второй переводит курсор на следующую строку.

Зная особенности клавиши Enter, легко понять, почему условие продолжения ввода выглядит в нашей программе как

```
cmp   cRead. 2
jnz   NewLine
```


Ведь cRead равное 2, говорит о нажатии только клавиши Enter, то есть, по сути, об окончании ввода. Значит, ввод имеет смысл продолжить, когда cRead больше двух.

Тут, правда, есть одна тонкость. Оказывается, процедура ReadConsole откликается еще на одну комбинацию клавиш Ctrl+Z (нажав и удерживая Ctrl, нажимаем Z). Эта комбинация срабатывает в любой момент, даже до нажатия Enter, и приводит к тому, что ReadConsole прекращает работу, а число прочитанных символов становится равным нулю. По своему смыслу клавиши Ctrl+Z должны прерывать выполнение программы, но у нас они не срабатывают, потому что условный переход jnz NewLine произойдет, когда разность cRead - 2 не равна нулю, то есть и после нажатия Ctrl+Z. Чтобы клавиши Ctrl+Z прерывали выполнение программы, нужно изменить условие:

```
cmp   cRead. 2
ja    NewLine
```

Условный переход ja NewLine означает «переход, если больше» (с буквы «а» начинается английское слово *after — после*).

Под «больше» процессор понимает такое состояние флагов, когда флаг нуля Z = 0 (результат вычитания cRead - 2 ненулевой) и флаг переноса C опущен. Легко убедиться, что процессор прав. Действительно, при cRead больше двух результат сравнения cmp cRead. 2 положителен — значит, заема из старшего разряда не возникло и флаг переноса опущен (см. раздел «Сложение и вычитание»). Естественно, при положительном результате сравнения опущен и флаг нуля, поэтому условие выполняется, и программа переходит к вводу следующих символов.

Если же cRead меньше двух, при сравнении cRead и 2 возникает заем из старшего разряда, поднимается флаг

переноса, переход не выполняется и программа завершает работу.

Кроме инструкций jnz и ja процессор понимает множество других инструкций перехода. Немного зная английский, можно догадаться о названиях многих из них по двум уже известным. Раз есть переход по неравенству нулю (jnz), то должен быть и переход по равенству (jz). Кроме перехода по «больше» должен быть переход по «меньше» jb (b — первая буква слова *below*, то есть *ниже*). Что такое «меньше», легко понять, зная, что такое «больше». Переход по «меньше» случится, когда флаг нуля опущен, а флаг переноса, наоборот, поднят.

Естественно, должны быть переходы по «больше или равно» — jae ('e' — первая буква слова *equal*, то есть *равный*). Аналогично jbe — переход, когда «меньше или равно».

Поскольку сам процессор не различает положительные и отрицательные числа, можно догадаться, что для чисел со знаком и без знака нужны особенные инструкции перехода. Инструкции jb и ja, с которыми мы только что познакомились, работают с числами *без знака*. Для чисел со знаком предназначены инструкции jg ('g' — начальная буква слова *greater — больше*) и jl ('l' — начальная буква слова *lower — меньше*). Основные инструкции перехода по результатам сравнения чисел со знаком и без знака показаны в таблице 5.1.

Таблица 5.1. Переходы после инструкции сравнения cmp

Числа без знака		Числа со знаком	
Инструкция	Переход, если...	Инструкция	Переход, если...
JA	C = 0 и Z = 0	**JG**	Z = 0 и S = O
JBE	C = 1 или Z = 1	**JLE**	Z=1 или S ≠ O
JB	C = 1	**JL**	S ≠ O
JAE	C = 0	**JGE**	S = O

Условия перехода для чисел со знаком могут показаться странными: почему, например, переход по «меньше» jl наступает, когда флаг знака S не равен флагу переполнения (в табл. 5.1 нужно отличать нули от буквы O)? Но давайте разберем несколько примеров сравнения, чтобы убедиться в том, что процессор и на этот раз прав.

Пусть сравниваются два отрицательных числа: –3 и –5

```
mov  ax, -3
mov  bx, -5
cmp ax, bx
```

Вычитая из –3 число –5 получим –3 – (–5) = –3 + + 5 = 2. Очевидно, флаг знака будет опущен, ведь результат положительный. Но опустится и флаг переполнения, потому что при вычитании из –3 (его дополнительный код FFFD) –5 (дополнительный код FFFB) не возникает заема. Итак, флаги знака и переполнения равны, и команда jg совершит переход к указанной метке. Точно так же поведет себя в этом случае и команда ja, потому что число –3 представляется большим двоичным числом, чем –5.

Пусть теперь сравниваются отрицательное число –3 и положительное 5.

```
mov  ax, -3
mov  bx, 5
cmp ax, bx
```

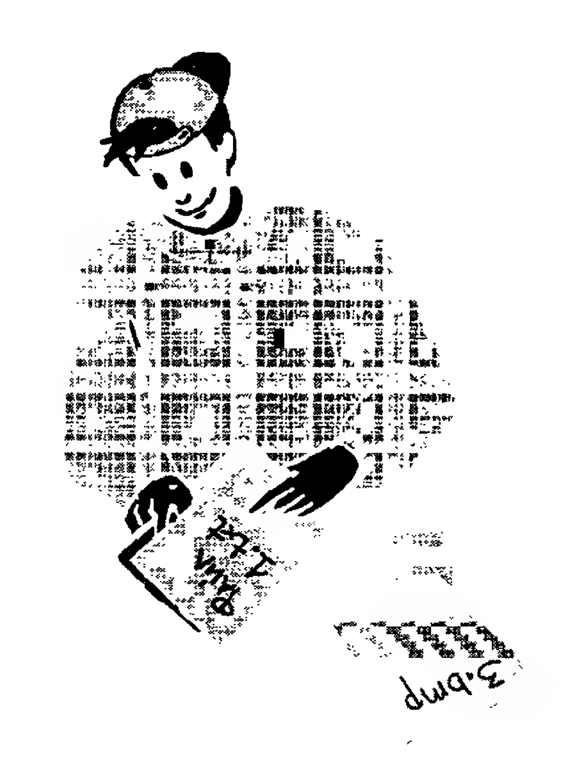
Сравнение cmp состоит в том, что флаги процессора устанавливаются так, как будто из ax вычли bx, при этом сам регистр ax не меняется. Что же будет при таком вычитании? Когда из –3 вычитают 5, получится число –8, то есть флаг знака поднимется (результат отрицательный), а флаг переполнения O опустится, потому что никакого переполнения нет. С точки зрения сравнения чисел со знаком возникло событие «меньше»

и команда jl отбросит процессор к указанной метке. Но с точки зрения сравнения чисел без знака –3 больше 5, потому что дополнительный код числа –3 это FFFD, то есть 65533, что гораздо больше 5.

Мораль: необходимо очень тщательно выбирать инструкции перехода. Неправильная инструкция очень коварна, потому что во многих случаях ведет себя так, как ожидалось. Между тем, никто, кроме программиста не может знать, какие числа (со знаком или без знака) сравниваются. Для процессора все числа равны.

Кроме переходов, зависящих от результатов сравнения, есть переходы, обусловленные состоянием флагов: jc (переход при поднятом флаге переноса) и, соответственно, jnc — переход при опущенном флаге. Есть переходы по флагу переполнения jo и знака js. Не стоит стараться запомнить все инструкции. Достаточно знать основные их типы и иметь под рукой хороший справочник.

ГЛАВА 6 Файлы



Открытие файла

Находя простые числа в главе 4, мы не задумывались о том, где будем их хранить, потому что все они умещались на экране монитора. Но представим себе, что нужно получить не десять, а *десять тысяч* простых чисел. Для такой задачи экран монитора будет маловат и нужно подумать о *файлах*, в огромных количествах хранящихся на жестком диске компьютера.

В файлах можно хранить все: тексты книг, изображения, музыку, программы и, конечно же, числа, поскольку все это можно представить длинной последовательностью нулей и единиц.

Программа, показанная в листинге 6.1, сохраняет четыре числа 3, 5, 7, 11 в файле simple.

Листинг 6.1. Сохранение четырех чисел в файле

```
.386
.model  flat, stdcall
option  casemap:none
include    \myasm\include\windows.inc
include    \myasm\include\kernel32.inc
includelib \myasm\lib\user32.lib
includelib \myasm\lib\kernel32.lib
NOFDIG equ 4
DSIZE  equ 4
BSIZE  equ NOFDIG*DSIZE
.data
fName    db "simple",0
fHandle  dd ?
cWritten dd ?
digs     dd 3,5,7,11
.code
start:
invoke CreateFile, ADDR fName,
       GENERIC_WRITE,
       0, NULL, CREATE_ALWAYS,
       FILE_ATTRIBUTE_ARCHIVE, 0
mov    fHandle, eax
invoke WriteFile, fHandle, ADDR digs,BSIZE,
       ADDR cWritten, NULL
invoke CloseHandle, fHandle
invoke ExitProcess, 0
end start
```

Работа с файлом начинается с его создания или открытия (если файл уже существует). Всем этим управляют многочисленные параметры процедуры CreateFile, с которыми стоит познакомиться подробнее.

Первый параметр содержит адрес имени файла, состоящего из символов. Признаком окончания имени служит нулевой байт. В нашем случае этот параметр равен ADDR fName — адресу нулевого символа в массиве fName, который состоит из шести символов simple и завершающего нуля.

Второй параметр показывает процедуре, *для чего* открывается или создается файл. Параметр GENERIC_WRITE означает, что разрешена запись в файл, GENERIC_READ разрешает только чтение. Можно позволить и чтение и запись. Для этого оба параметра объединяются оператором ИЛИ:

```
GENERIC_READ or GENERIC_WRITE
```

Третий параметр показывает, может ли файл использоваться другими программами (не забывайте, что Windows — многозадачная операционная система!). Нас пока многозадачность не интересует, поэтому выбираем параметр, равный нулю, что открывает доступ к файлу только нашей программе.

Четвертый параметр тоже имеет отношение к многозадачности. Он содержит адрес области данных,

в которой указано, может ли файл использоваться программами, порожденными данной. Любая программа в системе Windows может запускать другие программы или, как говорят, *процессы*. И эти процессы могут иметь доступ к файлу, созданному «родительским» процессом. Мы пока не доросли до запуска «дочерних» программ, поэтому полагаем этот параметр равным нулевому адресу NULL, что разрешает использовать файл только основной программе.

Пятый параметр показывает, что делать, если файл уже существует. Значение CREATE_ALWAYS приказывает уничтожить уже существующий файл и создать на его месте пустой файл с тем же именем. В нашем случае это значит, что если в папке, где запущена программа, нет файла simple, то он будет создан. Если же такой файл уже есть, он будет уничтожен, и на его месте появится файл с тем же именем simple, но совершенно пустой. Могут пригодиться и другие значения этого параметра: CREATE_NEW (не трогает уже существующий файл), OPEN_EXISTING (открывает только уже существующий файл, сохраняя его содержимое), OPEN_ALWAYS (если файл существует, открывает его, не трогая содержимое. Если не существует, создаем новый файл с указанным именем).

Шестой параметр задает атрибут файла: архивный, скрытый, только для чтения, системный и т. д. В нашем случае параметр равен FILE_ATTRIBUTE_ARCHIVE — обычному для большинства файлов атрибуту.

Едва ли нам стоит что-то знать о седьмом параметре, ведь число возможных комбинаций предыдущих шести равно, по некоторым оценкам, 33554432. Поэтому будем считать, что он всегда равен нулю.

Как бы ни сочетались между собой эти семь параметров, процедура CreateFile возвращает всего одно число в регистре eax. Это *дескриптор* файла, используемый аналогично дескриптору экрана или клавиатуры.

Например, процедура WriteFile, используемая для записи в файл символов (см. листинг 6.1), имеет те же параметры, что и процедура WriteConsole: дескриптор файла fHandle, адрес области памяти, хранящей символы, — ADDR digs, количество символов BSIZE, адрес числа, где хранится число действительно записанных в файл символов ADDR cWritten, и, наконец, никому не нужный нулевой указатель NULL. Процедура WriteFile пишет в файл 16 байт — таков суммарный размер четырех двойных слов, где размещены простые числа 3, 5, 7, 11.

Завершает работу с файлом процедура CloseHandle, у которой всего один параметр — дескриптор файла. Эта процедура отсоединяет файл от дескриптора, и после ее выполнения файл снова должен быть открыт, чтобы стали возможными чтение или запись в файл.

Прежде чем перейти к следующему разделу, обратим внимание на директиву

```
BSIZE  equ NOFDIG*DSIZE,
```

которая задает число записываемых символов. Оказывается, ассемблер способен не только заменять имя соответствующим числом, но и выполнять простейшие арифметические действия. В результате появится новое имя BSIZE, которое ассемблер заменит в тексте программы числом 16.

Чтение

Программа, написанная в предыдущем разделе, создает рядом с собой на диске файл simple, где записаны четыре числа. Полезно заглянуть внутрь этого файла, для чего в оболочке FAR служит кнопка F3. Подсветив имя

файла и нажав F3, увидим содержимое файла, показанное символами. Но поскольку в нашем файле хранятся числа, полезнее увидеть соответствующие этим символам шестнадцатеричные коды. Для этого после F3 следует нажать F4, и тогда нам откроется примерно то же, что на рис. 6.1.

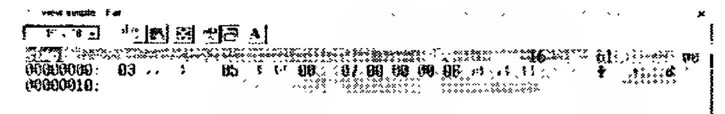


Рис. 6.1. Внутренности файла simple

В файле, как и в памяти компьютера, числа выворачиваются наизнанку: младшие биты идут первыми. Каждое число занимает четыре байта, причем номера байтов даны в шестнадцатеричной системе: адрес 10 соответствует десятичному числу 16. Такого байта в нашем файле нет, ведь их нумерация начинается с нуля и все 16 байтов имеют номера от 0 до 15.

Научившись создавать файлы и записывать туда числа, подумаем о том, как их читать. Весь наш предыдущий опыт подсказывает, что для этого нужно сначала получить дескриптор файла (для этого подойдет процедура CreateFile), а затем использовать процедуру чтения, которая должна быть похожей на процедуру записи. Программа, читающая только что созданный файл и выводящая хранящиеся в нем числа на экран, показана в листинге 6.2.

Листинг 6.2. Чтение файла и вывод его содержимого на экран


```
.386
.model  flat, stdcall
option  casemap:none
include     \myasm\include\windows.inc
include     \myasm\include\user32.inc
include     \myasm\include\kernel32.inc
includelib \myasm\lib\user32.lib
includelib \myasm\lib\kernel32.lib
NOFDIG equ 4
DSIZE  equ 4
BSIZE  equ NOFDIG*DSIZE
DIGSZ  equ 10
.data
fName     db "simple",0
fmt       db "%d",0
fHandle   dd ?
stdout    dd ?
cRead     dd ?
cWritten dd ?
buf       dd BSIZE dup (?)
dig2sim   db DIGSZ dup (?)
.code
start:
invoke  GetStdHandle, STD_OUTPUT_HANDLE
mov stdout, eax
invoke CreateFile, ADDR fName,\
       GENERIC_READ,\
       0, NULL, OPEN_EXISTING,\
       FILE_ATTRIBUTE_NORMAL, 0
mov    fHandle, eax
invoke ReadFile, fHandle, ADDR buf,\
       BSIZE, ADDR cRead, NULL
mov ecx, NOFDIG
mov esi, 0
nxt:
push ecx
invoke  wsprintf, ADDR dig2sim,\
        ADDR fmt, buf[esi]
invoke  WriteConsole, stdout,\
        ADDR dig2sim,\
        DIGSZ, ADDR cWritten, NULL
add esi,4
pop ecx
loop nxt
invoke CloseHandle, fHandle
invoke ExitProcess, 0
end start
```

Процедура CreateFile используется в ней не для создания, а для открытия файла. Файл, который мы собираемся читать, уже существует, поэтому используется параметр OPEN_EXISTING, запрещающий процедуре заново создавать файл с указанным именем.

С уже существующим файлом следует обращаться осторожно, чтобы ненароком не уничтожить хранящиеся там данные. Поэтому используется параметр GENERIC_READ, указывающий процедуре, что файл открыт только для чтения.

После открытия файла и запоминания его дескриптора за дело принимается процедура чтения файла ReadFile, устроенная так же, как и WriteFile. Ее параметры: дескриптор файла fHandle, адрес буфера, где окажутся прочитанные символы, число символов BSIZE, адрес переменной, хранящей число на самом деле прочитанных символов: ADDR cRead и, наконец, завершающий NULL, нам уже знакомы. Заметим только, что количество на самом деле прочитанных символов не всегда равно указанному. Больше символов, чем есть в файле, прочитать нельзя. Сравнивая число прочитанных и число указанных символов, можно понять, что достигнут конец файла. Обычно файл читают в несколько приемов, каждый раз сравнивая число заданных и число на самом деле прочитанных символов. Если оба числа равны, конец файла не достигнут и чтение продолжается. Если же прочитано меньше символов, чем указано, файл пришел к концу и чтение завершается.

Остаток программы из листинга 6.2 должен быть нам понятен. В нем добытые из файла цифры выводятся на экран. Делается это в цикле

```
nxt:
push ecx
```



```
add esi. 4
pop ecx
loop nxt
```

Сначала число из массива преобразуется в символы с помощью wsprintf, а затем выводится на экран процедурой WriteConsole. Поскольку процедуры Windows API уничтожают переменную цикла ecx, приходится сохранять ее в стеке и восстанавливать инструкцией pop ecx перед каждым новым оборотом цикла. Регистр esi хранит относительный адрес числа в массиве buf. Каждый раз он увеличивается на 4, потому что таков размер числа в байтах. Регистр esi не нужно сохранять в стеке, потому что об этом уже заботятся все процедуры Windows API (см. раздел «Повторение» главы 4).

Программа из листинга 6.2, с которой мы только что познакомились, содержит, как и всякая другая, по крайней мере одну ошибку. Хорошо, что эта ошибка очевидна и ее легко исправить. Дело в том, что, открывая файл процедурой CreateFile, мы были непростительно беспечны, считая, что файл с указанным именем действительно существует. Но представим себе, что это не так. Как поведет себя процедура, пытаясь открыть несуществующий файл, — пока не ясно. А между тем в описании процедуры CreateFile сказано, что возвращаемое значение будет в этом случае отличаться от всех возможных «нормальных» дескрипторов файлов. Это значение равно INVALID_HANDLE_VALUE, и, открывая файл, нужно всякий раз проверять, не равен ли ему полученный дескриптор.

Задача 6.1. Перепишите программу из листинга 6.2 с учетом возможных ошибок процедуры CreateFile. Испытайте программу неправильным именем файла и убедитесь в том, что она и в этом случае ведет себя разумно.

Интернет — источник знаний

Существует, как мы уже знаем, порядка полутора тысяч процедур Windows API. И в такой небольшой книге, как эта, невозможно рассказать даже о малой их части. Впрочем, едва ли стоит это делать в любой, пусть даже очень толстой книге. Гораздо удобнее использовать Интернет или справочные программы, которые можно найти в том же Интернете.

Чтобы, например, узнать подробнее о процедуре `CreateFile`, достаточно соединиться с поисковой системой Google (`www.google.com`), набрать в поле поиска слово «createfile» (Google не различает строчные и прописные буквы) и уже первый результат поиска откроет нам справочник по Windows API на сайте msdn.microsoft.com (рис. 6.2).

В правой части окна браузера видна статья о процедуре `CreateFile`, а в левой — ссылки на другие процедуры API. Вся документация написана на «родном» английском языке, но для ее понимания достаточно выучить 200–300 слов[1]. Посмотрим же, как выглядит описание уже известной нам функции в «фирменной» документации.

Для тех, кто уже знаком с процедурой, достаточно посмотреть на ее прототип, чтобы вспомнить, как ею пользоваться:

```
HANDLE CreateFile(
LPCTSTR lpFileName,
DWORD dwDesiredAccess,
DWORD dwShareMode,
LPSECURITY_ATTRIBUTES lpSecurityAttributes,
DWORD dwCreationDisposition,
DWORD dwFlagsAndAttributes,
HANDLE hTemplateFile
);
```

[1] Можно, конечно, найти материалы по Windows API и на русском языке. Но все равно большая часть документации написана на английском, и программисту никак без него не прожить. Чем раньше вы начнете учить английский, тем лучше.

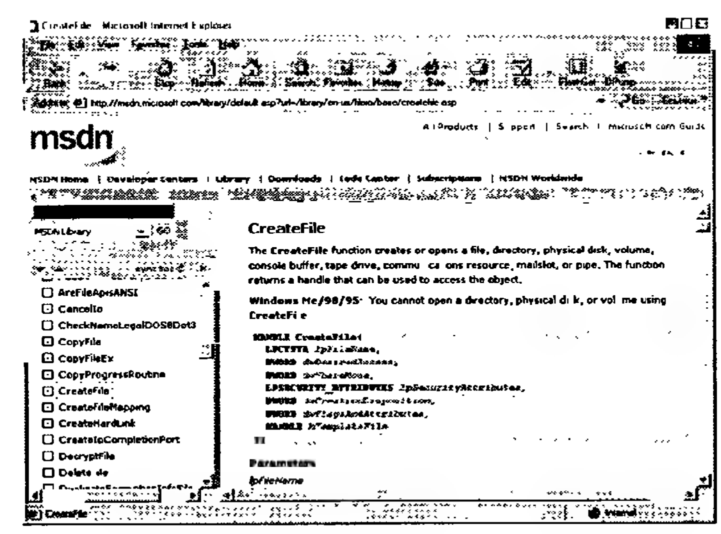


Рис. 6.2. Подробные сведения о процедуре CreateFile

Описание процедуры начинается типом возвращаемого значения, затем идет ее имя и далее в круглых скобках — список параметров:

```
HANDLE CreateFile(j);
```

В нашем случае процедура `CreateFile` возвращает значение типа `HANDLE` — так обозначается в документации фирмы Microsoft дескриптор файла. Мы уже знаем, что это целое число, хранимое в регистре eax. Но слово `HANDLE` подсказывает нам, как это значение будет использоваться.

Сами параметры процедуры указаны в той последовательности, в какой мы записываем их при ее вызове директивой invoke. Каждый параметр представлен двумя словами: первым идет название типа, вторым — образец имени переменной. Эти имена содержат массу полезной информации, которая открывается только знающим английский язык. Например, имя первого параметра lpFileName ясно говорит нам, что он связан с именем файла. Ведь *FileName* — и есть в переводе с английского *имя файла*. Буквы lp означают *long pointer*, то есть *длинный указатель*. Под словом *указатель* понимается адрес, то есть результат действия операторов offset или ADDR. Что же касается слова *long* (*длинный*), то все адреса в Windows *длинные*. О *коротких* адресах мы узнаем в главе 8.

После адреса нулевого элемента массива, хранящего имя файла, в описании функции идут два параметра, имеющих тип DWORD, то есть *double word*, двойное слово или, проще говоря, 4 байта. Имя параметра dwDesiredAccess означает в переводе с английского *желаемый доступ*, а префикс dw говорит о том, что это *двойное слово*. Остальные параметры разбираются аналогично.

Кроме справочников, расположенных на веб-сайтах, можно найти и целый файл в формате .hlp («родном» формате справочных файлов Windows). Размер его громаден и даже в сжатом виде он занимает около 8 Мбайт. Но лучше переписать его один раз, чем тратить время и деньги на блуждания по сайтам. Файл с описанием процедур Windows API называется win32api.zip и скачать его можно, например, здесь:

http://win32assembly.online.fr/files/win32api.zip

Командная строка

Задавать имя читаемого файла в исходном тексте программы, как мы это делали в разделе «Чтение», крайне неудобно. Нужно либо все время называть файл одним именем, либо каждый раз заново компилировать программу. Поэтому консольными приложениями управляют, помещая необходимые имена и параметры в командной строке. Если, например, программе нужно передать имя файла, то его пишут справа от ее имени.

Как выглядит в оболочке FAR передача параметра simple программе cl2.exe, находящейся в папке F:\asmtest\files, показано на рис. 6.3. При запуске программы клавишей Enter ей будет передан адрес командной строки; узнать его поможет специальная процедура GetCommandLine.

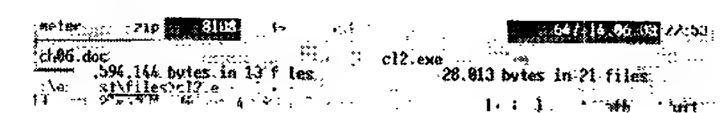


Рис. 6.3. Командная строка программы

Пользоваться этой процедурой учит программа из листинга 6.3, выводящая на экран свою собственную командную строку.

Листинг 6.3. Вывод собственной командной строки

```
.386
.model  flat, stdcall
option  casemap:none
include    \myasm\include\windows.inc
include    \myasm\include\kernel32.inc
includelib \myasm\lib\user32.lib
includelib \myasm\lib\kernel32.lib
LLNG  equ 128
```

продолжение ⇨

Листинг 6.3 *(продолжение)*

```
.data
stdout   dd ? ;дескриптор экрана
cWritten dd ? ;число показанных символов
CLIni    dd ? ;начало командной строки
.code
start:
invoke  GetStdHandle, STD_OUTPUT_HANDLE
mov stdout, eax
invoke GetCommandLine  ;адрес командной
                       ;строки
mov CLIni, eax         ;запомнить адрес
                       ;командной строки
mov edi, eax    ;edi – нач. ком. стр.
cld             ;будем увеличивать адрес
mov ecx, LLNG   ;макс. дл. ком. строки
mov al, 0       ;ищем 0
repne  scasb    ;поиск нуля
sub edi, CLIni  ;edi = длине строки
invoke  WriteConsole,  stdout,
    CLIni, edi, ADDR cWritten, NULL
invoke  ExitProcess, 0
end start
```

Как видим, процедура GetCommandLine не имеет параметров, а результат ее работы — адрес начала командной строки — оказывается в регистре eax.

Полезно рассмотреть подробнее эту командную строку с помощью отладчика. Для этого перейдем в папку, где расположена программа (будем считать, что ее имя 163.exe), вызовем отладчик командой

```
Ollydbg 163.exe simple
```

и «прокрутим» программу на несколько шагов вперед, так чтобы в регистре eax оказался адрес командной строки. Рисунок 6.4 как раз показывает такой момент.

Как только в регистре eax оказывается адрес 817BCCE8, отладчик справа от него показывает строку

символов, которая начинается с этого адреса. В нашем случае строка такова:

```
""f:\asmtest\files\163.exe" simple"
```

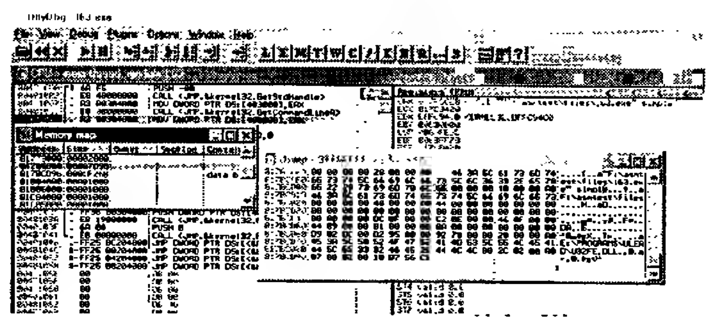


Рис. 6.4. Командная строка в памяти компьютера

В нее, как видим, попадает не только имя программы, но и путь к ней. Важно понять, как отладчик узнает о конце строки. Для этого изучим так называемый *дамп* памяти, то есть шестнадцатеричные коды и соответствующие им символы, находящиеся вблизи адреса 817BCCE8. Чтобы получить дамп памяти в нашем отладчике, выберем раздел меню View и далее — пункт Memory, после чего возникнет окно Memory map (показано в левой части рис. 6.4). В этом окне видны адреса участков памяти, доступных программе. Среди них нужно двойным щелчком мыши выбрать ближайший к тому, что нас интересует, и на экране появится окно Dump, видное в правой части рис. 6.4. Там показаны адреса и соответствующие им байты, представленные символами и шестнадцатеричными кодами. Выбрав символ мышью, увидим соответствующий шестнадцатеричный код, выделенный серым цветом. На рисунке выделена последняя буква слова simple. Ее шестнадцатеричный код

равен 65. А следом за ним идет ноль — 00. Вот что помогает отладчику правильно показать командную строку!

Воспользуемся и мы этим свойством, чтобы самим вывести командную строку на экран. Поскольку адрес начала строки нам известен, остается только найти адрес ее конца, что позволит узнать ее длину и вызвать затем процедуру WriteConsole.

Чтобы найти нулевой символ, достаточно сравнить с нулем все символы, начиная с адреса, выданного процедурой GetCommandLine. Если символ равен нулю, поиск окончен, если нет — адрес увеличивается на единицу и сравнение повторяется.

В программе из листинга 6.3 эти сравнения выполняет специальная инструкция scasb. Буква «b» в конце ее имени показывает, что сравниваются байты, и одновременно наводит на мысль, что можно сравнивать простые, 2-байтовые (scasw) и двойные (scasd) слова.

Команда scasb сравнивает байт, чей адрес находится в регистре edi, с байтом из регистра al. Результат сравнения показывает флаг нуля Z, а регистр edi, независимо от результата сравнения, увеличивается или уменьшается на единицу. В какую сторону будет меняться edi, зависит от специального, еще неизвестного нам *флага направления* D (рис. 6.5).

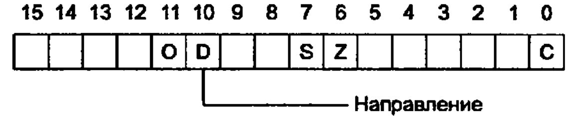


Рис. 6.5. Флаг направления

Когда флаг опущен, проверка идет в сторону увеличения адресов, когда поднят, — в сторону уменьшения. Чтобы задать направление поиска, существуют специальные команды, поднимающие (std) или опускающие (cld) флаг переноса. В нашей программе (см. листинг 6.3) флаг опущен, задавая тем самым движение в сторону увеличения адреса.

Инструкция scasb проверяет текущий байт и увеличивает edi. Чтобы искать с ее помощью нулевой символ, используется префикс repne, велящий инструкции scasb повторяться до тех пор, пока текущий байт, адрес которого находится в регистре edi, не станет равен тому, что хранится в регистре al, или пока не станет равным нулю регистр ecx. В регистре ecx задается максимальная дистанция поиска, у нас она выбрана равной 128.

Как видим, поиск с помощью scasb может прекратиться по двум причинам:

- достигнут конец строки, но заданный символ так и не найден;
- символ найден внутри строки.

Чтобы убедиться в том, что символ найден, достаточно проверить после инструкций repne scasb — поднят ли флаг Z. Если поднят, символ найден, если опущен — достигнут конец строки, но символа все нет:

```
repne   scasb       ;поиск символа
jnz     not_found   ;символ не найден
```

В программе из листинга 6.3 такой проверки нет, потому что мы знаем: нулевой символ обязательно будет найден. Вместо проверки jne not_found программа сразу находит длину строки, вычитая из текущего значения edi адрес начала строки, запомненный в переменной CLIni. Для проверки — не ошиблись ли мы на единицу, представим себе строку, состоящую из одного нулевого символа. Очевидно, повторение инструкции scasb прекратится уже на этом символе. Но scasb устроена так, что всегда

«перелетает» на позицию вперед или назад (в зависимости от направления движения). Когда поднимается флаг Z, scasb все равно увеличивает edi и лишь тогда прекращает работу. Поэтому в edi после окончания поиска будет адрес не завершающего нуля, а идущего следом за ним символа. Значит, вычитая из текущего значения edi адрес начала строки CLIni, то есть адрес нулевого символа, получим единицу, что и требовалось. Дальнейшее просто. После нахождения длины вызывается процедура WriteConsole, которая показывает командную строку на экране.

Kiss-принцип

В предыдущем разделе мы вычисляли длину командной строки, казалось бы, оптимальным способом: вычитая из текущего значения edi запомненный адрес начала строки:

```
sub edi, CLIni  ;edi - длине строки.
```

Но ассемблер — очень хитрый язык, располагающий к разным фокусам и трюкам. Чтобы показать, что запоминание начала строки излишне, вспомним, что поиск нулевого элемента гарантирован. Значит, можно сделать ecx сколь угодно большим и не бояться, что процессор будет искать нулевой элемент вечно. Итак, решено: сделаем ecx максимальным, то есть установим все его биты в единицу. Теперь во время поиска ecx будет уменьшаться на единицу и для достижения нуля можно проделать более 4 миллиардов сравнений. Но нулевой элемент найдется гораздо раньше. Спрашивается: как по значению ecx найти длину строки? Очевидно, из начального (самого большого) значения нужно вычесть конечное, ведь scasb «проскакивает» на шаг вперед. Выходит, нужно все-

таки запомнить начальное значение? А вот и нет! Вспомним, что число можно считать и положительным и отрицательным — в зависимости от того, что нам удобнее. Какому числу соответствуют все биты, установленные в единицу? Очевидно, это –1. После проверки нулевого элемента строки ecx уменьшится на единицу и станет равен –2, затем –3 и т. д. Пусть, например, строка состоит из одного символа 0, стоящего в ее начале. Тогда ecx после операции поиска будет равен –2. Выходит, для того чтобы вычислить длину строки, нужно обратить знак ecx и вычесть единицу. Изменение знака выполняет в ассемблере команда neg. Значит, инструкции, оставляющие длину строки в ecx, будут такими:

```
push   edi
cld
mov    ecx, -1
mov    al, 0
repne  scasb
neg    ecx
dec    ecx
pop    edi
```

Перед поиском нуля приходится сохранять в стеке адрес начала строки edi, необходимый процедуре WriteConsole, ведь инструкция scasb «портит» edi.

Только что полученный фрагмент программы можно «улучшить», если вспомнить, что изменение знака связано с инвертированием (заменой всех единиц нулями и всех нулей единицами) всех битов в регистре и прибавлением единицы (см. раздел «Знак» главы 2). Инвертирование в ассемблере выполняет инструкция not. Значит, инструкцию neg можно заменить последовательностью

```
;neg ecx
not ecx
inc ecx
```

Но в нашем отрывке следом за neg ecx идет инструкция dec ecx, уничтожающая эффект inc ecx. Выходит, пара инструкций neg ecx dec ecx заменяется одной — not ecx.

Окончательные инструкции поиска выглядят странно:

```
push   edi
cld
mov    ecx, -1
mov    al, 0
repne  scasb
not    ecx
pop    edi,
```

но опытные программисты легко их поймут.

Вообще, программистам следует держаться KISS-принципа (KISS — первые буквы английских слов **Keep It Simple Stupid** — делай проще, дурачок!). Чем скучнее написана программа, тем лучше — прежде всего самому ее автору. Ведь то, что кажется «крутым» и остроумным сейчас, через две недели станет просто непонятным. Но ассемблер — особый язык, и границы непонятного в нем размыты. Любой приличный программист поймет и оценит трюк с вычислением длины строки по изменению ecx. Попытки нестандартного решения задачи кроме вреда приносят еще и пользу, потому что помогают «сродниться» с языком.

Кроме того, изощренное программирование на ассемблере способно заставить программу, написанную на языке высокого уровня[1], выполняться быстрее. Современные процессоры очень мощны, но и задачи, ими решаемые, становятся все сложнее. Поэтому быстродействия процессора всегда не хватает. Вот здесь и пригождается ассемблер. Как правило, большую часть

[1] К языкам высокого уровня относятся C, C++, Pascal, Basic — словом, все, что не ассемблер.

времени занимает выполнение небольшого числа инструкций, создаваемых компилятором при переводе языка высокого уровня на ассемблер. Качество «перевода», как правило, не очень высоко. Поэтому имеет смысл переписать эти немногие инструкции вручную. И здесь понятность отходит на второй план. Главной становится быстрота выполнения. Вот почему строгое следование KISS-принципу не так важно в ассемблере, как в языках более высокого уровня.

Открытие файла — 2

Теперь, наконец, все готово к тому, чтобы «изъять» имя файла из командной строки и открыть его цивилизованно, не касаясь исходного текста программы.

Сделать это просто, зная, что имя файла находится в самом конце командной строки. Поэтому переместимся в ее конец с помощью инструкции scasb, затем изменим направление поиска и найдем с помощью scasb уже пробел. Символы, расположенные между пробелом и концом командной строки (нулевым символом) — и есть, очевидно, имя файла.

Процедура, возвращающая адрес первого символа имени файла в регистре eax, показана в листинге 6.4.

Листинг 6.4. Получение адреса первого символа имени файла

```
GetFName proc
invoke GetCommandLine
mov   edi, eax  ;зап. адр. нач. ком. строки
mov   ecx, -1
cld             ;поиск вперед
mov   al, 0     ;ищем ноль
repne scasb
dec   edi       ;промахнулись — назад
mov   cx, -1
```

продолжение ⇒

Листинг 6.4 *(продолжение)*

```
std             ;ищем в обратном напр.
mov   al, 32    ;пробел
repne scasb
cmp   cx, -3    ;есть имя?
jz    empty     ;нет - сообщаем
add   edi, 2    ;да - прыгаем через пробел
mov   eax, edi  ;возвращаем адрес
ret
empty:
mov   eax, -1   ; нет имени
ret
GetFName endp
```

Сначала вызывается GetCommandLine, чтобы получить адрес командной строки. Затем инструкцией cld задается поиск в сторону увеличения адресов. После того как ноль найден, необходимо вернуться назад (dec edi), потому что инструкция scasb проскакивает мимо нуля. Далее направление поиска меняется на противоположное и ищется уже пробел (его код равен 32_{10} или 20_{16}). После инструкции scasb необходимо проверить, есть ли какие-нибудь символы между пробелом и завершающим нулем. Если ecx изменился всего на 2 (cmp ecx, -3), то символов никаких нет, а значит, в командной строке имя файла не указано. В этом случае процедура возвращает −1. Если же ecx изменился больше, то между нулем и пробелом есть хотя бы один символ. Это и есть имя файла. Чтобы получить адрес первой его буквы, нужно увеличить edi на 2. Мы ведь искали пробел, но scasb промахивается на шаг, поэтому edi после завершения поиска указывает на символ, стоящий левее пробела. Чтобы регистр edi указывал на пробел, нужно к нему прибавить единицу. Прибавив еще единицу, мы перескочим через пробел — как раз к первому символу имени файла.

Создав процедуру GetFName, легко написать и всю программу, которая пробует открыть указанный файл, сообщая только о неудачах: отсутствии файла в текущей папке или о том, что имени файла вообще нет в командной строке (листинг 6.5).

Листинг 6.5. Открытие файла

```
.386
.model  flat, stdcall
option  casemap:none
include    \myasm\include\windows.inc
include    \myasm\include\kernel32.inc
includelib \myasm\lib\user32.lib
includelib \myasm\lib\kernel32.lib
GetFName proto
.data
fHandle  dd ?
stdout   dd ?
cWritten dd ?
error    db "Нет такого файла"
noname   db "Укажите имя файла"
.code
start:
main proc
invoke  GetStdHandle, STD_OUTPUT_HANDLE
mov stdout, eax
invoke GetFName
cmp    eax, -1
jz     empty
invoke CreateFile, eax,
       GENERIC_READ,
       0, NULL, OPEN_EXISTING,
       FILE_ATTRIBUTE_NORMAL, 0
       cmp eax,INVALID_HANDLE_VALUE
       jz  exit
mov    fHandle, eax
invoke CloseHandle, fHandle
invoke  ExitProcess, 0
exit:
```

продолжение

всегда разлучены. Им ничего не известно друг о друге.

«Локальность» меток очень важна, потому что позволяет не думать о стандартных именах. Например, в процедурах часто есть метка, куда отправляется процессор в случае ошибки или, наоборот, благополучного завершения процедуры. Первую можно все время называть error (*ошибка*), вторую — exit (*выход*), не беспокоясь о том, что в соседней процедуре они названы так же.

Но все-таки иногда нужно преодолеть локальность метки, сделав ее доступной всем процедурам. Для этого ее имя дополняется справа парой двоеточий:

```
global::  ;метка, доступная всем процедурам
```

Прогулки по файлу

До сих пор мы читали и записывали файлы целиком — от начала до самого конца. Но так бывает далеко не всегда. Иногда необходимо пропустить начало файла или записать что-то в его середину. Чтобы проделать такое, нужно понимать, что файл *читается последовательно*. Прочитать десятый байт можно только переместив к нему специальный указатель. Этот указатель автоматически перемещается при чтении-записи файла, поэтому для чтения десятого байта можно прочитать предыдущие девять. А можно ничего не читать, а просто переместить указатель с помощью процедуры SetFilePointer.

Посмотрим, как работает эта процедура на примере редактирования файла cook, содержащего фразу

```
ПЕРЧИТЬ НЕЛЬЗЯ СОЛИТЬ
```

В этой фразе пропущена запятая, от правильного положения которой, зависит судьба блюда. Будем считать,

называть error (*ошибка*), вторую — exit (*выход*), не беспокоясь о том, что в соседней процедуре они названы так же.

Но все-таки иногда нужно преодолеть локальность метки, сделав ее доступной всем процедурам. Для этого ее имя дополняется справа парой двоеточий:

```
global::  ;метка, доступная всем процедурам
```

Прогулки по файлу

До сих пор мы читали и записывали файлы целиком — от начала до самого конца. Но так бывает далеко не всегда. Иногда необходимо пропустить начало файла или записать что-то в его середину. Чтобы проделать такое, нужно понимать, что файл *читается последовательно*. Прочитать десятый байт можно, только переместив к нему специальный указатель. Этот указатель автоматически перемещается при чтении/записи файла, поэтому для чтения десятого байта можно прочитать предыдущие девять. А можно ничего не читать, а просто переместить указатель с помощью процедуры SetFilePointer.

Посмотрим, как работает эта процедура на примере редактирования файла cook, содержащего фразу

```
ПЕРЧИТЬ НЕЛЬЗЯ СОЛИТЬ
```

В этой фразе пропущена запятая, от правильного положения которой зависит судьба блюда. Будем считать, что запятая должна стоять после перчить. Тогда нашу задачу решает программа, показанная в листинге 6.6.

Листинг 6.6. Редактирование файла

```
.386
.model  flat, stdcall
option  casemap:none
include    \myasm\include\windows.inc
```

продолжение ⇒

Листинг 6.6 *(продолжение)*

```
include    \myasm\include\kernel32.inc
includelib \myasm\lib\user32.lib
includelib \myasm\lib\kernel32.lib
BSIZE  equ 128
.data
fName    db "cook",0
fHandle  dd ?
cRead    dd ?
cWrite   dd ?
cWritten dd ?
comma    db ","
buf      db BSIZE dup (?)
.code
start:
invoke CreateFile, ADDR fName,
       GENERIC_READ+GENERIC_WRITE,
       0, NULL, OPEN_EXISTING,
       FILE_ATTRIBUTE_NORMAL, 0
mov    fHandle, eax
invoke SetFilePointer, fHandle, 7,
       NULL,FILE_BEGIN
invoke ReadFile, fHandle, ADDR buf,
       BSIZE, ADDR cRead, NULL
invoke SetFilePointer, fHandle, 7,
       NULL,FILE_BEGIN
invoke WriteFile, fHandle, ADDR comma,
       1, ADDR cWrite, NULL
invoke WriteFile, fHandle, ADDR buf,
       cRead, ADDR cWrite, NULL
invoke CloseHandle, fHandle
invoke  ExitProcess, 0
end start
```

Все начинается в ней с привычного уже открытия файла процедурой CreateFile. Но теперь мы решились *редактировать* файл, поэтому приходится открывать его на чтение и запись, для чего комбинируются два параметра:

```
GENERIC_READ+GENERIC_WRITE
```

Запомнив дескриптор файла, займемся перемещением указателя процедурой SetFilePointer. Этот указатель можно представить себе как флажок, помечающий соответствующий байт, но для процессора существуют только числа. Поэтому позиция указателя — тоже число со знаком, хранящееся в двойном слове. Отрицательное число означает перемещение указателя назад, положительное — вперед. Чтобы перемещение стало однозначным, нужно задать точку отсчета. Для процедуры SetFilePointer их существует три: FILE_BEGIN (от начала файла), FILE_END (от конца) и FILE_CURRENT (от текущей позиции указателя).

Зная все это, легко догадаться, что вызов процедуры

```
invoke SetFilePointer, fHandle, 7,
       NULL, FILE_BEGIN
```

означает перемещение указателя на 7 байтов вперед относительно начала файла. Отсчет байтов в файле начинается с нуля, поэтому нулевое положение указателя соответствует букве П, первое — букве Е, а указатель, равный семи, помечает пробел, стоящий сразу за словом ПЕРЧИТЬ. Именно туда нужно вставить запятую, но прежде необходимо прочитать и сохранить остаток фразы. Этим занимается процедура ReadFile:

```
invoke ReadFile, fHandle, ADDR buf,
       BSIZE, ADDR cRead, NULL
```

Символы, начиная с пробела и кончая словом СОЛИТЬ, читаются в массив buf. Их число BSIZE задано заведомо большим, все равно прочитать больше символов, чем есть, нельзя, и верное их число сохранится в переменной cRead. После чтения «хвоста» файла указатель перемещается в самый его конец и показывает на несуществующий символ, стоящий непосредственно за мягким знаком в слове СОЛИТЬ.

Но нам необходимо поставить запятую сразу за словом ПЕРЧИТЬ, поэтому вновь вызывается процедура SetFilePointer, перемещающая указатель на семь позиций вперед относительно начала файла. Если теперь записать один символ в файл, то он встанет туда, куда показывает указатель, то есть на место пробела, стоящего за словом ПЕРЧИТЬ. В нашей программе процедура WriteFile пишет туда запятую, после чего указатель продвигается на шаг вперед и теперь у него восьмая позиция относительно начала файла. Чтобы закончить редактирование, достаточно записать в файл сохраненный в массиве buf фрагмент, что и делает последняя инструкция WriteFile.

Задача 6.2. Как записать в файл запятую, задавая ее положение не от начала файла, а от текущей позиции указателя файла?

ГЛАВА 7 Дроби



Надо держаться корней

Находя простые числа в главе 4, мы использовали самый тупой из всех возможных алгоритмов: делили каждое число-кандидат N на все числа от 2 до N – 1, и если ни одно из них не делилось нацело, справедливо считали число N простым.

Между тем почти половина делений была заведомо напрасной, потому что делить на числа, превышающие N/2, не имеет смысла, и если, скажем, при испытании числа 17 деления на числа от 2 до 8 не дали нулевого остатка, то деление на числа от 9 до 16 можно не проводить.

Но и это не предел. Оказывается, прекращать деление можно при достижении целочисленного значения ON. Ведь если число делится на корень из себя, то в нем два равных сомножителя N = ON * ON. Если же делить на число, большее чем ON, то второй сомножитель (в случае деления нацело) будет уже меньше, чем ON. Но ведь, проверяя число на «простоту», мы уже поделили его на числа, меньшие ON, и нашли, что таких нет! Значит, деление на числа, *большие* ON, бессмысленно. Возьмем, например, число 17. Целочисленное значение корня из 17 равно 4. Проверяя числа 2, 3, 4, найдем, что 17 на них не делится. Но проверять число 5 смысла уже не имеет, потому что второй сомножитель будет заведомо меньше четырех (5 * 4 равно уже 20), а такие мы уже проверяли.

Итак, для нахождения простых чисел (и для множества других задач) необходимо вычислять корни из чисел, а поскольку они далеко не всегда целые, нужно еще уметь представить их последовательностью нулей и единиц, потому что ничего другого в компьютере просто нет.

Эта задача легко решается, если сообразить, что степени двойки, применяемые в двоичном коде, могут быть не только положительными, нулевыми, но и *отрицательными*. Договорившись, где в регистре находится граница между положительными и отрицательными степенями двойки, можно хранить там дробные величины. Если предположить, что в 8-битовом регистре точка разделяет тетрады (старшие и младшие четверки бит), то число 11111111 будет равно

$$2^3 + 2^2 + 2^1 + 2^0 + 2^{-1} + 2^{-2} + 2^{-3} + 2^{-4} = 8 + 4 + 2 + 1 + 1/2 + 1/4 + 1/8 + 1/16 = 15{,}9375$$

Так кодируются *числа с фиксированной точкой*. В реальности, конечно, используется гораздо большее число бит, но все равно их не хватает для хранения огромных чисел, легко возникающих при делении какого-то большого числа на очень маленькое. Вот почему дроби часто представляются в виде произведения числа с фиксированной точкой (мантиссы) на множитель, равный двойке в какой-то степени. Степень двойки называют экспонентой и хранят в виде обычного двоичного числа без знака. Кроме мантиссы и экспоненты нужен еще и бит, кодирующий знак числа. Все три составляющие (знак, мантисса и экспонента) занимают непрерывный участок памяти и составляют вместе *число с плавающей точкой*, которое может храниться в 32, 64 или 80 битах. «Плавать» точку заставляет экспонента: ведь умножение мантиссы на степень двойки как раз и соответствует смещению границы, отделяющей целую часть числа от дробной. На рис. 7.1 показано, как представлены в компьютере 32- и 64-битовые числа с плавающей точкой.

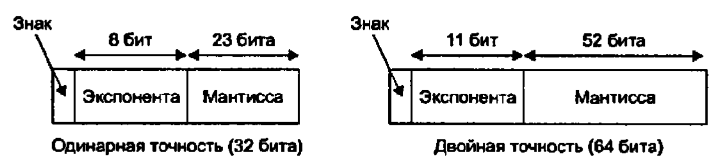


Рис. 7.1. Формат чисел с плавающей точкой

Задача 7.1. Оцените максимальное число десятичных знаков после запятой, а также диапазон чисел с одинарной и двойной точностью. Не забудьте, что экспонента может быть как положительной, так и отрицательной.

Как видим, числа с плавающей точкой довольно сложно устроены и к ним нельзя сразу применить обычные арифметические инструкции. Если бы мы вздумали складывать или умножать числа с плавающей точкой, пользуясь инструкциями mul, div, add, sub, то пришлось бы выделять мантиссу и экспоненту, произвести кучу вспомогательных действий и потом снова упаковать число в 32 или 64 бита.

Вот почему всю эту кропотливую, утомительную а, главное, требующую множества вычислений работу, берет на себя процессор. В нем, оказывается, есть специальные инструкции и регистры для обработки чисел с плавающей точкой. Некоторое представление о них дает программа, вычисляющая квадратный корень из числа 17 (см. листинг 7.1).

Листинг 7.1. Вычисление квадратного корня из числа

```
.386
.model  flat, stdcall
option  casemap:none
include    \myasm\include\windows.inc
include    \myasm\include\kernel32.inc
include    \myasm\include\fpu.inc
includelib \myasm\lib\user32.lib
includelib \myasm\lib\kernel32.lib
includelib \myasm\lib\fpu.lib
```



```
BSIZE equ  30
.data
sqroot     dt  ?
digit      dd  17
stdout     dd  ?
cWritten   dd  ?
buf   db BSIZE dup (?)
.code
start:
main proc
invoke  GetStdHandle, STD_OUTPUT_HANDLE
mov stdout, eax
fild digit       ;загружаем целое в регистр
fsqrt            ;вычисляем корень
fstp sqroot      ;сохраняем в 80 битах
invoke  FpuFLtoA, ADDR sqroot, 10, \
        ADDR buf, SRC1_REAL or SRC2_DIMM
invoke  WriteConsoleA, stdout, ADDR buf, \
        BSIZE, ADDR cWritten, NULL
invoke  ExitProcess, 0
main endp
end start
```

В «сердце» этой программы находятся три инструкции:

```
fild digit  ;загружаем целое в регистр
fsqrt       ;вычисляем корень
fstp sqroot ;сохраняем в 80 битах
```

загружающие целое число 17 в специальный регистр (fild digit), вычисляющие корень (fsqrt) и сохраняющие результат в 80 битах под именем sqroot (fstp sqroot).

Полученный корень затем выводится на экран процедурой FpuFLtoA, которая может работать только с 80-битовыми числами. Эта процедура входит в специальную библиотеку fpu.lib, подключаемую, как и остальные библиотеки, в начале нашей программы.

У процедуры FpuFLtoA четыре параметра: адрес отображаемого числа (ADDR sqroot), количество десятичных знаков после запятой (у нас — 10), адрес буфера, где

окажутся символы, в которые превратится число, и, наконец, константы, управляющие работой процедуры. Константа SRC1_REAL говорит функции, что ее первый параметр — это адрес 80-битового числа, хранящегося в обычной памяти. Обратите внимание на директиву dt: sqroot dt ? Так в ассемблере объявляется 10-байтовая переменная (с буквы «t» начинается английское слово *ten — десять*). Константа SRC2_DIMM указывает функции, что второй ее параметр — просто число. В нашем случае это 10. Раз существуют такие константы, разумно предположить, что первый и второй параметры процедуры могут быть другими, но об этом поговорим чуть позже.

А сейчас будет полезно подсмотреть за программой с помощью отладчика. На рис. 7.2 показано состояние регистров после выполнения команды загрузки числа fild digit.

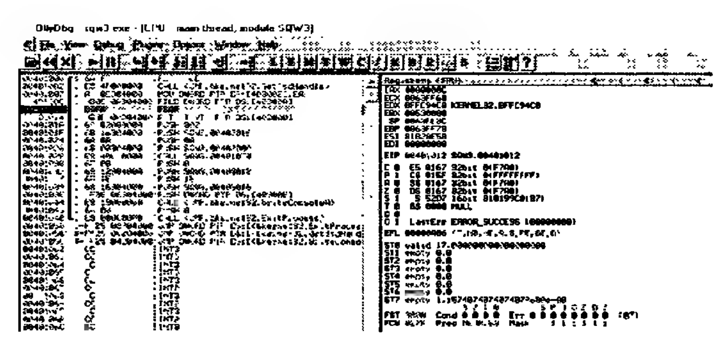


Рис. 7.2. Регистры, хранящие числа с плавающей точкой

Как видите, инструкция fild загружает целое число в один из регистров, хранящих числа с плавающей точкой. Всего в процессоре 8 таких регистров, носящих имена ST0, ST1,..., ST7. Число 17 оказывается в регистре ST0 (рис. 7.2, правый нижний угол.). Перед загрузкой оно преобразуется в специальный формат с плавающей точкой и занимает уже не 4, а 10 байтов — таков размер регистров ST0-ST7. В сущности, это совсем

другое число, потому отладчик и пишет 17.000000*, а не просто 17.

После загрузки числа наступает черед инструкции fsqrt, извлекающей из него корень, который занимает место самого числа в регистре ST0. Наконец, третья команда fstp sqroot переписывает корень из регистра ST0 в обычную 10-байтовую область памяти. Затем его «подхватывает» процедура FpuFLtoA, расшифровывает и записывает в буфер последовательность символов 4.1231056256 с десятью, как указано, знаками после запятой. А уж как работает процедура WriteConsole, мы знаем.

Процессор и сопроцессор

Мы такие разные, но все-таки мы вместе!

Рекламный слоган

Регистры и команды процессора, ответственные за «перемалывание» чисел с плавающей точкой, столь отличны от других команд и регистров процессора, что будет лучше говорить о них как об отдельном устройстве, называемом *сопроцессором*. Давным-давно, когда трудно было уместить все в одной микросхеме, это и были отдельные устройства, работавшие независимо друг от друга. Программисту приходилось даже использовать команды ожидания wait и fwait, чтобы «притормозить» одно устройство, когда ему необходимы были результаты работы другого. Эта независимость сохранилась и сейчас, когда «такие разные» процессор и сопроцессор расположились на одном кристалле. Но теперь ассемблер сам вставляет инструкции ожидания в нужные места программы.

Чем же так отличаются процессор и сопроцессор? Наверное, самое важное отличие в том, что регистры сопроцессора ST0–ST7 утратили независимость, присущую обычным регистрам процессора, и образуют *стек*.

Загружаемое в сопроцессор число попадает на вершину стека, при этом числа, уже хранящиеся в других регистрах, смещаются на шаг от вершины. В стеке могут храниться 8 чисел — столько, сколько в нем регистров. Попытка загрузить в стек девятое число, приведет к потере числа, далее всего отстоящего от вершины. Но и вершина при этом не воспримет то, что в нее загружается, и будет содержать некое значение, которое с точки зрения сопроцессора не может быть числом. На рис. 7.3 показано состояние регистров сопроцессора после загрузки девяти чисел 1, 2, 3...9.

```
ST0 bad   -NAN FFFF C0000000 00000000
ST1 valid 8.0000000000000000000
ST2 valid 7.0000000000000000000
ST3 valid 6.0000000000000000000
ST4 valid 5.0000000000000000000
ST5 valid 4.0000000000000000000
ST6 valid 3.0000000000000000000
ST7 valid 2.0000000000000000000
```

Рис. 7.3. Сопроцессор хранит только 8 чисел

Первым в сопроцессоре оказалось число 1.0. Оно заняло вершину стека, то есть регистр ST0. Далее на вершину стека попало загруженное вторым число 2.0, а число 1.0 спустилось ниже — в регистр ST1. Затем на вершине стека побывали числа 3.0, 4.0, 5.0, 6.0, 7.0, 8.0. Число 1.0, попавшее в стек первым, спускалось все ниже и оказалось, наконец, в регистре ST7, когда на его вершине было число 8.0. Но при попытке запихнуть в стек девятое число случилась авария: единица, загруженная первой, покинула стек, а на вершине оказалось неверное значение, помеченное словом bad (в переводе с английского *плохой*). Кроме «плохих» в стеке могут быть нормальные числа, помеченные словом valid. Таковы все числа, видные на рисунке, кроме первого. У регистров ST0–ST7 может быть еще один атрибут empty. Так помечается регистр,

в который можно загрузить число. Если регистр занят, то его нужно перед использованием освободить. Делается это инструкцией ffree. Чтобы, например, освободить третий регистр, нужна инструкция ffree ST(3). Есть еще одна инструкция finit, которая освобождает все регистры и чаще всего используется для приведения стека в некое исходное состояние, от которого удобно «плясать».

Знакомясь с устройством сопроцессора, читатель, наверное, не раз уже говорил себе: «почему, по какой причине сопроцессор устроен так странно, так непохоже на обычный процессор, работающий хоть и с целыми, но тоже числами»? Чтобы ответить на этот вопрос, попробуем вычислить с помощью сопроцессора разность произведений

alpha*beta - delta*gamma

Листинг 7.2. Сопроцессор вычисляет разность произведений

```
.386
.model  flat, stdcall
option  casemap:none
include    \myasm\include\windows.inc
include    \myasm\include\kernel32.inc
include    \myasm\include\fpu.inc
includelib \myasm\lib\user32.lib
includelib \myasm\lib\kernel32.lib
includelib \myasm\lib\fpu.lib
BSIZE equ  30
.data
alpha        dd  0.5
beta         dd  1.37
gamma        dd  220.0
delta       dd  0.65
stdout      dd  ?
cWritten    dd  ?
buf         db BSIZE dup (?)
.code
start:
main proc
finit
fld alpha
fld beta
```

продолжение ↗

Листинг 7.2 *(продолжение)*

```
fmul
fld gamma
fld delta
fmul
fsub
invoke  GetStdHandle. STD_OUTPUT_HANDLE
mov stdout. eax
invoke  FpuFLtoA. 0. 10. ADDR buf.
        SRC1_FPU or SRC2_DIMM
Invoke  WriteConsoleA. stdout. ADDR buf. \
        BSIZE. ADDR cWritten. NULL
invoke  ExitProcess. 0
main endp
end start
```

Программа, показанная в листинге 7.2, сначала инициализирует сопроцессор инструкцией finit. Затем помещает в стек с помощью команд fld два первых сомножителя

```
fld alpha
fld beta
```

После загрузки в стек число alpha окажется в регистре ST1, a beta — на вершине стека в регистре ST0. Теперь настает черед инструкции fmul, умножающей ST1 на ST0, помещающей результат умножения в ST1 и затем выталкивающей из стека значение beta, оставшееся на вершине. Иными словами, после инструкции fmul на вершине стека окажется произведение alpha * beta, а сами значения alpha и beta, более нам не нужные, покинут сопроцессор.

Теперь можно загрузить вторую пару сомножителей

```
fld gamma
fld delta
```

после чего на вершине стека окажется delta, в регистре ST1 — gamma, а в регистре ST2 — произведение alpha * beta, которое вытесняется к окраинам стека, но не теряется, и после второй инструкции fmul на вершине окажется произведение delta * gamma, а в регистре ST1 — произведение alpha * beta. Легко догадаться, что следующая инструкция fsub вычтет из регистра ST1 содержимое регистра ST0 и поместит результат этой операции на вершину стека в регистр ST0.

Как видим, стековая организация сопроцессора очень удобна для вычислений, потому что пара операндов, загнанная в стек, естественно заменяется результатом действия над ней. А сам результат легко сохраняется в стеке и может участвовать в следующих действиях. Для регистров, образующих стек, идеальна так называемая *обратная польская запись*, когда сначала идут операнды, а следом за ними — знаки действий. Наша сумма произведений запишется на обратный польский манер следующим образом:

```
alpha beta * gamma delta * -
```

Сопроцессору очень легко понять такую запись: каждое имя переменной означает помещение в стек, а каждый знак действия говорит о том, что берутся два операнда (один — из вершины стека, другой — ближайший к ней), и результат действия, вытесняя один из операндов, оказывается на вершине.

По сути программа из листинга 7.2 как раз и использует такую запись, полученную интуитивно, вручную. Но есть специальные процедуры, которые автоматически преобразуют формулы в обратную польскую запись, поступающую на вход сопроцессора.

Слово состояния

Но не всегда вычисления проходят так гладко. Иногда нужно оставить один из операндов в стеке или изменить порядок действий (например, вычислить разность ST0 - ST1) или же использовать операнд, хранимый вдалеке от вершины. Чтобы все это стало возможным, команды сопроцессора используют явно заданные аргу-

менты, причем один из них обязательно должен быть вершиной стека. Например, инструкция

```
fsub ST(3). ST
```

вычисляет разность ST(3) - ST(0) (вместо ST0 можно писать просто ST), помещает результат в ST(3) и при этом ничего не делает со стеком. Чтобы инструкция, чьи аргументы указаны явно, освобождала вершину стека, ей необходим суффикс p:

```
fsubp ST(3). ST ;ST(3) = ST(3) - ST(0)и pop
```

В инструкциях возможен и один операнд, например fsub digit. Такая инструкция понимается сопроцессором как команда вычесть из вершины стека число digit, которое может занимать 4 или 8 байт обычной памяти. Результат оказывается на вершине стека. Заметим, что ассемблер не примет суффикс p в команде fsubp digit, потому что вычисление разности и немедленное ее удаление из стека — операция бессмысленная, даже для такого «тупицы», как сопроцессор.

Чтобы понять, как работают разные команды сопроцессора, попробуем найти корни квадратного уравнения $x^2 + px + q$, вычисляемые по формулам:

```
root1 = -(p/2) + 0(p²/4 - q)
root2 = -(p/2) - 0(p²/4 - q)
```

Программа, показанная в листинге 7.3, решает квадратное уравнение при p = –6, q = 5. Как будет меняться состояние стека после выполнения инструкции сопроцессора, показано на рис. 7.4.

Листинг 7.3. Решение квадратного уравнения

```
.386
.model  flat. stdcall
option  casemap:none
include    \myasm\include\windows.inc
include    \myasm\include\kernel32.inc
include    \myasm\include\fpu.inc
includelib \myasm\lib\user32.lib
```



```
includelib \myasm\lib\kernel32.lib
includelib \myasm\lib\fpu.lib
BSIZE equ  30
.data
p           dd      -6.0
q           dd       5.0
two         dd      -2.0
root1       dt       ?
root2       dt       ?
stdout      dd       ?
cWritten    dd       ?
buf         db BSIZE dup (?)
.code
start:
finit
fld p:ST-p
fld  two  :ST - -2. ST(1) = p
fdiv      :ST(1) = p/2 и pop
fld ST    :ST(0) = ST(1) = p/2
fmul      :ST(1) - ST(1)*ST(0) - p²/4 и pop
fld q     :ST(0) = q
fsub      :ST(0) = (p²/4) - q
fsqrt     :√((p²/4) - q))
fld p     :ST(0) = p
fld two   :-2.0
fdiv      :ST(0) = -p/2
fld  ST   :ST(1) = ST(0)
fsub ST.ST(2)   :ST = -p/2 - √((p²/4) - q))
fstp root1      :сохранить корень
fadd ST.ST(1)   :ST = -p/2 + √((p²/4) - q))
fstp root2      :сохранить корень
invoke  GetStdHandle. STD_OUTPUT_HANDLE
mov  stdout. eax
invoke  FpuFLtoA. ADDR root1. 10. \
        ADDR buf. SRC1_REAL or SRC2_DIMM
invoke  WriteConsoleA. stdout. ADDR buf. \
        BSIZE. ADDR cWritten. NULL
invoke  FpuFLtoA. ADDR root2. 10. \
        ADDR buf. SRC1_REAL or SRC2_DIMM
invoke  WriteConsoleA. stdout. ADDR buf. \
        BSIZE. ADDR cWritten. NULL
invoke  ExitProcess. 0
end start
```

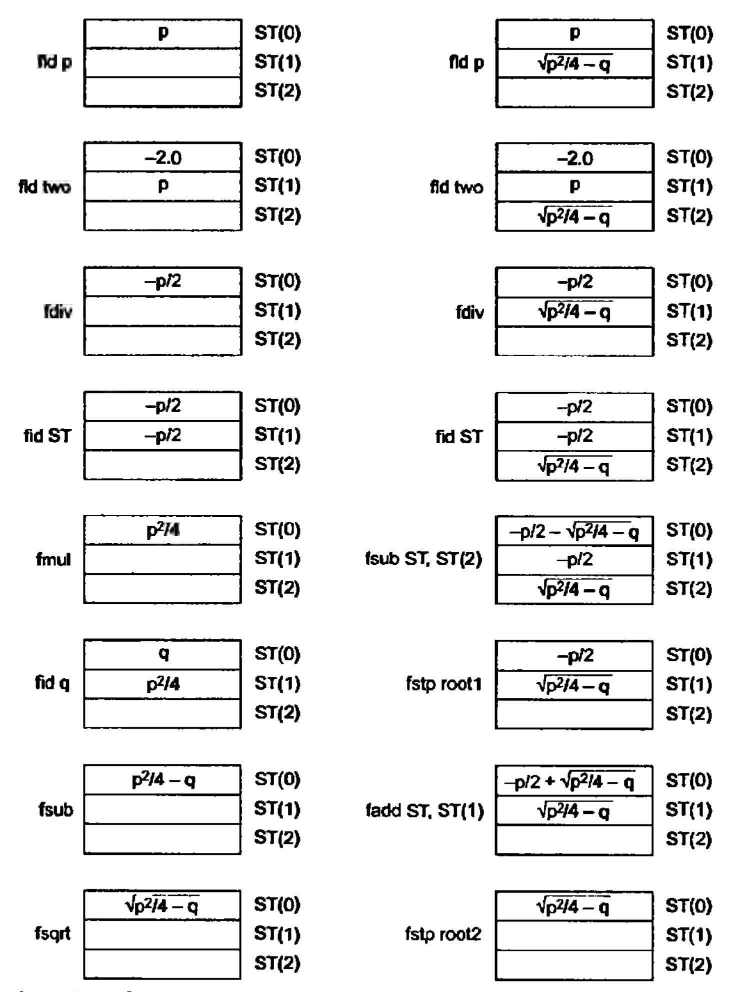


Рис. 7.4. Состояние регистров после команд сопроцессора

Все эти инструкции мы уже неплохо знаем, за исключением fdiv, по умолчанию делящей ST(1) на ST, и, быть может, fld ST, просто копирующей вершину стека.

Программа из листинга 7.3, несмотря на свой приличный размер, никак не защищается от отрицательного значения $(p^2/4) - q$, которое получается при отсутствии действительных корней уравнения. Как поведет себя сопроцессор при попытке вычислить корень из отрицательного числа, мы пока не знаем. Но ясно, что ничего хорошего из этого не выйдет.

Поэтому нужны инструкции, проверяющие значения в регистрах сопроцессора, подобно обычным инструкциям test и cmp. В сопроцессоре такая инструкция называется ftst. Не имея аргументов, она просто сравнивает вершину стека с нулем. Результат сравнения хранится в 3 битах C2, C1, C0 специального слова состояния сопроцессора (рис. 7.5).

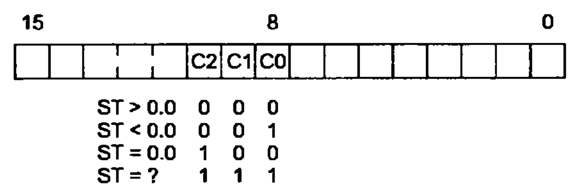


Рис. 7.5. Слово состояния сопроцессора и возможные результаты проверки инструкцией ftst

Как видим, отрицательное или неверное значение вершины стека получается при единичном бите C0. Чтобы проверить этот бит, достаточно прочитать слово состояния в обычное 2-байтовое слово и затем проверить младший бит старшего байта. Все это проделывают инструкции, показанные в листинге 7.4.

Листинг 7.4. Проверка вершины стека

```
fsubp ST(1),ST
ftst          ;проверить вершину стека
fstsw ax      ;прочитать слово состояния
shr ah, 1     ;C0 -> флаг переноса
jc    exit    ;Если число < 0 – выход
fsqrt         ;нет – вычисляем корень
```

Инструкция ftst проверяет слово на вершине стека, а инструкция fstsw ax переписывает слово состояния, содержащее результат проверки в регистр ax. Сдвиг регистра ah на шаг вправо shr ah, 1 помещает восьмой бит слова состояния во флаг переноса, а инструкция jc exit отправляет процессор к метке exit, когда этот флаг поднят. Если же флаг опущен, число на вершине стека не отрицательно и к нему применима операция извлечения корня fsqrt.

Модульность

Возьмемся за руки, друзья, чтоб не пропасть поодиночке.

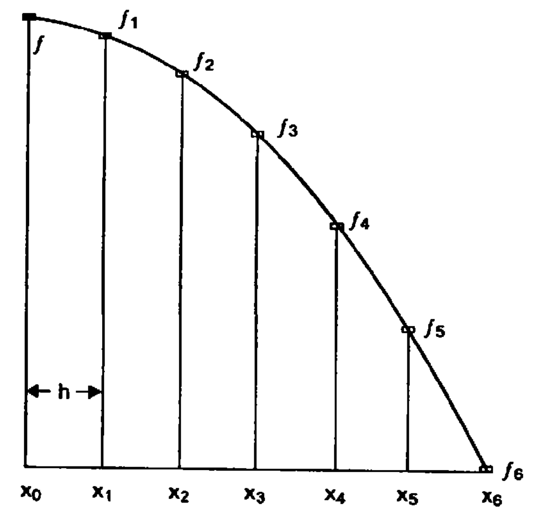
Б. Окуджава. «Союз друзей»

Небольшую программу, занимающую один-два экрана монитора, удобно хранить в одном файле. Там ее легко охватить взглядом и как угодно менять, компилировать, запускать на исполнение и снова менять. Наши прежние программы были именно такими.

Но представим себе программу даже не из тысяч, а из нескольких сотен строк, хранящуюся в одном файле. Чтобы ее отладить, неизбежно придется перемещаться из одного конца файла в другой. И будет трудно удержать в памяти увиденное в начале программы, спеша к ее концу.

Сложность программы, содержащей множество дублирующих, мешающих друг другу переменных и функций, растет столь стремительно, что уже при длине в несколько сотен строк *она* начинает управлять программистом, а не он ею. Чтобы удержать контроль над сложностью, такую программу следует разбить на несколько как можно более независимых частей, которым, в отличие от друзей Окуджавы, необходимо быть поодиночке, чтобы не пропасть.

Поясним сказанное примером вычисления интеграла от функции f(x) с помощью формулы Симпсона, использующей значения функции, взятые в 2n + 1 фиксированных точках $f(x_0)$, $f(x_1)$, $f(x_2)$ ...,$f(x_{2n})$. Веса, приписываемые значениям функции, различны: нулевое и последнее значения берутся с весом 1, нечетные значения (x_1, x_3, x_5...) имеют вес 2, четные (x_2, x_4, x_6)— вес 4. Если n равно 3, то функция вычисляется в семи точках (x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7) и формула Симпсона получается такой, как на рис. 7.6.



$$\int f(x)dx = \frac{h}{3}\left[f_0 + 4f_1 + 2f_2 + 4f_3 + 2f_4 + 4f_5 + f_6\right]$$

Рис. 7.6. Формула Симпсона при n = 3

Теперь можно написать процедуру вычисления интеграла. Чтобы она была универсальной, значения функции при соответствующих x_i будет находить другая процедура Fun, которая просто возьмет число с вершины

стека и заменит его значением функции. То, что получилось после примерно 10-й попытки, показано в листинге 7.5

Листинг 7.5. Процедура simpson.asm

```
.386
.model  flat, stdcall
option  casemap:none
Fun     PROTO
.data
two     dq 2.0
three   dq 3.0
four    dq 4.0
.code
Simpson  proc X0:DWORD,X2N:DWORD,NN:DWORD,
              H:QWORD,SUMADDR:DWORD
LOCAL    dstep:QWORD
Fld     H        ;загрузить шаг
fld     two      ;загрузить 2.0
fmul             ;шаг*2
fst     dstep    ;сохранить двойной шаг
fldz             ;загрузить сумму
fld     X0       ;начало интервала
fld     H        ;шаг
fadd             ;x0 + step
mov     ecx,NN   ;число слагаемых
fld     ST       ;дублируем x0 + step
nxt:   invoke Fun ;вычисляем функцию
faddp  ST(2), ST  ;суммируем + pop
fadd   ST, ST(2)  ;добавляем dstep
fld    ST         ;копируем новый x
loop   nxt        ;след. слагаемое
fcompp            ;убираем 2 числа
fld    four       ;ST = 4.0
fmul              ;sum = sum*4.0
fld    dstep      ;двойной шаг
fldz              ;sum = 0.0
fld    X0         ;
fld    dstep      ;
fadd              ;загружаем x0 + 2.0*H
mov ecx, NN
dec ecx           ;число слагаемых на 1 меньше
fld ST            ;дублируем x0 + dstep
nxt1:  invoke Fun ;вычисляем функцию
faddp  ST(2), ST  ;sum = sum + Fun(x) и pop
fadd   ST,ST(2)   ;x = x + dstep
fld    ST         ;дублируем x + dstep
loop   nxt1       ;новое слагаемое
fcompp            ;убираем два значения
fld    two        ;ST = 2.0
fmul              ;sum = sum * 2
fadd   ST, ST(2)  ;+предыдущая сумма
fld    X0
invoke Fun        ;fun(x0)
fadd              ;прибавим fun(x0)
fld    X2N        ;
invoke Fun        ;
fadd              ;прибавим fun(x2n)
fld    H          ;
fmul              ;sum = sum * h
fld    three      ;
fdivp  ST(1),ST   ;(h/3) * sum
mov    eax, SUMADDR
fstp   TBYTE PTR [eax];сохраняем интеграл
finit                 ;очищаем сопроцессор
ret
Simpson endp
end
```

Эта процедура очень похожа на программы, которые мы до сих пор писали, разница только в том, что после завершающей директивы end нет никакой метки. Эта метка (обычно мы называем ее start) должна быть только в одной главной программе, которую нам еще предстоит создать, а пока попробуем разобраться с тем, что есть в листинге 7.5.

Прежде всего посмотрим список параметров процедуры, всего их пять: X0 — начальное значение x, X2N — конечное значение x, NN — параметр n, определяющий число значений функции, по которым вычисляется интеграл. Таких значений в формуле Симпсона 2n + 1. Следующий параметр H — не что иное, как расстояние между соседними значениями x, например H = X1 - X0.

Этот параметр, часто называемый *шагом*, желательно задавать с большой точностью, ведь число точек, по которым вычисляется интеграл, может быть очень велико. Поэтому он занимает учетверенное слово или 8 байт (QWORD). И наконец, последний параметр SUMADDR — адрес в памяти, куда будет записан полученный интеграл. Этот адрес занимает, как обычно, двойное слово DWORD, то есть 4 байта.

За параметрами следуют данные. По сути это константы 2.0, 3.0, 4.0, необходимые для вычисления интеграла. Каждая константа задается с высокой точностью, занимает 8 байтов, то есть имеет тип QWORD и объявляется как dq, например,

```
three   dq 3.0 ;константа занимает 8 байтов
```

Сама процедура выглядит устрашающе, но стоит выделить в ней самые важные инструкции, обслуживанию которых подчинены все остальные, и окажется, что понять в ней нужно всего несколько строк.

Но прежде познакомимся с нехитрой идеей вычислений: общую сумму удобно разбить на четыре части: значение функции в начале интервала f(x0), в конце — f(x2n), сумма значений при нечетных x, умноженная на 4, сумма значений при четных x, умноженная на 2. После вычисления суммы ее еще нужно умножить на треть шага (H/3).

Очевидно, центральное место в процедуре занимают два цикла: первый вычисляет сумму значений при нечетных x, второй — соответственно сумму значений при четных. Оба цикла похожи, поэтому проследим только за работой первого:

```
mov   ecx, NN      ;число слагаемых
fld   ST           ;дублируем x0 + step
nxt:   invoke Fun  ;вычисляем функцию
faddp  ST(2), ST   ;суммируем + pop
fadd   ST, ST(2)   ;добавляем dstep
fld    ST          ;копируем новый x
loop   nxt         ;след. слагаемое
```


Чтобы хоть что-то понять в работе этого важнейшего участка процедуры, нужно проследить за инструкциями, которые ему предшествуют. А это, с учетом того, что мы уже знаем о сопроцессоре, нетрудно.

Перед запуском цикла в стек загружается двойной шаг ST(2) = 2H, начальное значение суммы ST(1) = 0.0 и первое значение x, в котором вычисляется функция ST(0) = x0 + H (рис. 7.7).

ST(0)	x0 + H
ST(1)	0.0
ST(2)	2H

Рис. 7.7. Стек сопроцессора перед первым оборотом цикла

Затем в ecx посылается число слагаемых mov ecx, NN (как видно из рис. 7.6), в формуле Симпсона NN слагаемых с весом 4 и NN-1 — с весом 2. И наконец, перед самым началом цикла дублируется вершина стека (fld ST). При этом начальное значение суммы окажется в ST(2). Цикл начинается вычислением значения функции invoke Fun, которое после вызова функции Fun окажется на вершине стека. Далее это значение прибавляется к сумме инструкцией faddp ST(2), ST и снимается со стека, потому что оно больше не понадобится. Теперь на вершине стека оказалось значение x, в котором только что вычислялось значение функции (вот для чего понадобилось копировать вершину стека!), и к нему следует прибавить двойной шаг, что и делает инструкция fadd ST,ST(2). Далее вершина стека снова копируется, и мы приходим к тому же состоянию стека, что и при первом обороте цикла. Разница лишь в том, что теперь на вершине и в ST(1) находится следующее значение x, при котором нужно вычислить функцию!

Оставшаяся часть процедуры, показанной в листинге 7.5, не требует пояснений. Пожалуй, стоит только сказать об инструкции fcompp, которая сравнивает два значения ST(0) и ST(1) и затем выталкивает их из стека. Мы используем эту инструкцию только для удаления лишних чисел, результат их сравнения нам не нужен.

Написав процедуру, можно переходить к основной программе, роль которой второстепенна: ей необходимо подготовить параметры, передаваемые процедуре, и вывести на экран значение интеграла. Попробуем вычислить простейший интеграл от функции cos(x) в пределах от нуля до $\pi/4$. Этот интеграл равен О2/2, и нам легко будет оценить точность его вычисления. Основная программа показана в листинге 7.6. Она не использует ничего нового и потому не нуждается в комментариях. Так что нам теперь осталось только сделать из подпрограммы и основной программы исполнимый файл с расширением .exe.

Листинг 7.6. Файл main.asm

```
.386
.model  flat, stdcall
option  casemap:none
include    \myasm\include\windows.inc
include    \myasm\include\kernel32.inc
include    \myasm\include\fpu.inc
includelib \myasm\lib\user32.lib
includelib \myasm\lib\kernel32.lib
includelib \myasm\lib\fpu.lib
Simpson    PROTO :DWORD, :DWORD, :DWORD,\
                 :QWORD, :DWORD
Fun        PROTO
N     equ  5
BSIZE equu 30
.data
buf          db BSIZE dup (?)
cWritten     dd ?
stdout       dd ?
iini         dd 0
iend         dd ?
n            dd N
two          dq 2.0
four         dq 4.0
step         dq ?
sum          dt ?
.code
start:
main proc
finit
fldpi              ;загрузить π
fld   four         ;ST = 4.0
fdiv               ;ST = π/4
fst   iend         ;сохранить iend
fld   iini         ;ST = iini
fsub               ;ST = iend - iini
fild n             ;ST - n
fld two             ;ST = 2.0
fmul               ;2 * n
fdiv               ;(iend - iini)/(2 * n)
fstp  step         ;сохранить шаг
invoke  Simpson, iini, iend, n, step,\
        ADDR sum
invoke  GetStdHandle, STD_OUTPUT_HANDLE
mov stdout, eax
invoke  FpuFLtoA, ADDR sum, 10, \
        ADDR buf, SRC1_REAL or SRC2_DIMM
invoke  WriteConsoleA, stdout, ADDR buf, \
        BSIZE, ADDR cWritten, NULL
invoke  ExitProcess, 0
main    endp
Fun proc
fcos
ret
Fun endp
end start
```

До сих пор мы не задумывались над загадочным превращением ассемблерного текста в объектный файл .obj и превращением объектного файла в исполнимый с расширением .exe. Пора понять, что объектные файлы нужны для подготовки отдельных частей программы к слиянию в один исполняемый файл.

В нашем случае нужно объединить программу main.asm (см. листинг 7.6) и подпрограмму simpson.asm (см. листинг 7.5), подготовив с помощью компилятора два объектных файла main.obj и simpson.obj и затем объединив их компоновщиком в один — main.exe. Для этого нам придется написать особый командный файл, показанный в листинге 7.7.

Листинг 7.7. Командный файл для создания программы из двух частей

```
ml /c /coff main.asm simpson.asm
link /SUBSYSTEM:CONSOLE main.obj simpson.obj
```

От уже привычного нам amake.bat он отличается тем, что компилирует сразу два файла main.asm и simpson.asm и затем объединяет в один исполнимый два объектных файла main.obj и simpson.obj.

Хранение программы в нескольких файлах позволяет не только управлять ее сложностью, но и многократно использовать отдельные ее части. Процедура simson.asm нарочно сделана независимой от основной программы, чтобы ее можно было использовать многократно.

Для этого пришлось заново объявить в процедуре simpson.asm константы two и four. Нужно отчетливо понимать, что two и four, объявленные в процедуре simpson.asm, — *совсем не те* two и four, что объявлены в main.asm. Компоновщик, объединяя объектные модули, заботится о том, чтобы two в процедуре simson.asm существовало отдельно от two в процедуре main и занимало совсем другой участок памяти.

Независимость, безусловно, хороша и следует к ней всячески стремиться. Но все же бывают случаи, когда процедурам суждено делить одни и те же данные. Например, регистр флагов у нас один, и переменную, в которой он хранится, второй раз не объявишь. Поэтому приходится применять разные уловки, чтобы процедуры пользовались одними и теми же данными.

Одна такая уловка использована при передаче значения интеграла процедуре main. Суть ее в том, что процедуре simpson.asm передается не само значение интеграла, которое еще предстоит вычислить, а его адрес: ADDR sum. Пользуясь косвенной адресацией, процедура записывает значение интеграла в 10-байтовую область памяти sum, определенную в процедуре main.asm:

```
mov     eax, SUMADDR
fstp    TBYTE PTR [eax] ;сохраняем интеграл
```

Кроме косвенной адресации сделать некую область памяти общей для разных процедур помогают директивы EXTERN и PUBLIC. Если область памяти общая, то и существовать она может только в единственном экземпляре, следовательно, объявить ее нужно только в *одной* из процедур. И там же нужно выделить ее директивой PUBLIC, чтобы показать ассемблеру, что переменная доступна другим процедурам. В других же процедурах, использующих эту переменную, нужна пометка EXTERN, которая просит компилятор «не волноваться» насчет этой переменной, мол, для нее уже выделена память, а где конкретно — определит компоновщик.

В качестве примера сделаем так, чтобы процедура simpson.asm использовала константы two и four, определенные в файле main.asm. Для этого нужно в файле main.asm пометить их как PUBLIC:

```
.386
.model  flat, stdcall
option  casemap:none
include    \myasm\include\windows.inc
...
includelib \myasm\lib\fpu.lib
public     two, four
Simpson      PROTO :DWORD, :DWORD, :DWORD,
           :QWORD, :DWORD
…
.data
```

```
...
two        dq 2.0
four       dq 4.0
...
```

А в файле simpson.asm нужно эти переменные пометить словом EXTERN:

```
.386
.model  flat, stdcall
option  casemap:none
Fun     PROTO
EXTERN  two:QWORD, four:QWORD
.data
three    dq 3.0
.code
...
```

Обратите внимание: теперь в процедуре simpson.asm память выделяется только одной переменной three dq 3.0, которая не помечена директивой PUBLIC и потому доступна только внутри файла simpson.asm. Память для переменных two и four не выделяется, потому что она уже выделена в процедуре main.asm. Об этом как раз и говорит директива EXTERN two:QWORD, four:QWORD. Встретив ее, компилятор поймет, как обращаться с переменными two, four, упомянутыми в файле simpson.asm. Директива EXTERN указывает имена переменных и их тип (в нашем случае это 8-байтовые слова QWORD) — это все, что нужно знать компилятору. А где выделить для них память, решит компоновщик. И в этом ему поможет директива PUBLIC.

ГЛАВА 8 16 бит



DOS

> Мы, как и люди, не живем вечно. Мы стареем, но стареют не тела наши, потому что им незнакомо понятие Время. Стареют исполняемые нами функции, становятся примитивными. И мы должны честно принять это и уйти сами, не дожидаясь, пока кто-то выпотрошит нас, высмеет и выбросит вон.
>
> *С. Расторгуев. «Программные методы защиты информации в компьютерах и сетях»*

Наверное, где-нибудь в пыльных углах еще можно разыскать компьютеры IBM PC-XT. Многие из них до сих пор исправны, только вряд ли кому придет в голову включать их, ведь современные операционные системы (такие, как Windows или Unix) нельзя на них запустить даже в принципе.

А ведь совсем недавно, в конце 80-х годов эти машины стоили бешеных денег и вызывали трепет у каждого настоящего программиста. Тогда в мире персональных компьютеров царила операционная система DOS (тоже фирмы Microsoft), которая управлялась командной строкой, примерно такой же, как в оболочке FAR. Сама эта оболочка тоже пришла к нам из тех времен. Хоть FAR и консольное приложение Windows, неспособное работать в системе DOS, он довольно точно копирует интерфейс оболочки Norton Commander, стоявшей в те годы на каждом компьютере.

В отличие от Windows DOS — *однозадачная* операционная система. Это значит, что она не способна одновременно выполнять несколько программ. То есть для передачи данных между программами приходится сначала сохранять их на диске, затем выходить из одной программы, запускать другую и читать сохраненные данные. Никакого буфера обмена в системе DOS нет. Его роль выполняют так называемые *резидентные* программы, которые «всплывают» при нажатии определенной комбинации клавиш, «сдирают» с экрана картинку или текст, сохраняют их в файле и передают управление предыдущей программе, которая уже может эти файлы прочитать.

Несмотря на все эти неудобства, DOS обладала и обладает важным достоинством: она дает полный контроль над компьютером, позволяет делать с ним и всеми его устройствами все что угодно. Программист (особенно если это умелый программист на ассемблере) чувствует, что может выжать из имеющегося «железа» все возможное и даже написать программу, способную уничтожить DOS, а вслед за ней и себя саму.

Блаженные времена, когда программист мог владеть целым компьютером, прошли. Современные операционные системы многое берут на себя: они не позволяют уже программам напрямую обращаться к устройствам компьютера, потому что программ несколько, а устройство — одно. Теперь программиста отделяет от «железа» толстый слой ваты — так называемый API (например, уже знакомый нам Windows API).

Но есть еще области (и немалые), где DOS может сослужить верную службу: это различные самодельные приборы, основанные на процессорах Intel Pentium. Такие приборы обычно собираются по частям, как конструктор: в специальную «корзину» вставляется материнская плата, а в нее — процессор, память и необходимые платы. Прибор обычно управляется единственной программой, которая должна взаимодействовать с нестан-

дартными устройствами, поэтому ее проще написать и удобнее выполнять в системе DOS.

Есть еще одна причина, по которой нужно быть знакомым с устройством программ для DOS: в мире осталось очень много исходных текстов на ассемблере для этой операционной системы. И чтобы не поддаться панике, увидев непонятные значки вроде int 21h, нужно познакомиться с DOS поближе.

Программированию на ассемблере для DOS посвящено множество книг. Поэтому мой рассказ коснется только самого главного. Но даже если вас не интересует DOS, эту и следующую главы все равно стоит прочитать. Потому что, говоря о DOS, мы узнаем много нового об инструкциях процессора и устройстве Windows.

А начнем с программы для DOS, выводящей на экран уже знакомую фразу Не могу молчать! (листинг 8.1).

Листинг 8.1. Не могу молчать! (DOS-версия)

```
.8086
.MODEL  small
.stack 100
.data
hello db  «Не могу молчать!». 0dh. 0ah. "$"
.code
start:
mov  dx.@stack
mov  ss.dx
mov  dx.@data :
mov  ds.dx    :регистр данных
mov  dx. offset hello
mov  ah. 09   ;вывести на экран
int  21h
mov  ah. 4ch  ;завершить программу
int  21h
end  start
```


Несмотря на многие новшества, вам должно быть в общих чертах понятно, что и как она делает. Так, например, строки

```
mov  ah. 09
int  21h
```

каким-то таинственным способом выводят на экран слова Не могу молчать!, а строки

```
mov  ah. 4ch
int  21h
```

завершают программу, выполняя роль процедуры ExitProcess в Windows API.

Программа, показанная в листинге 8.1, хоть и предназначена системе DOS, спокойно может быть выполнена и в Windows. В оболочке FAR она запускается так же, как и консольное приложение Windows, но если исследовать подробнее ее запуск и выполнение, то окажется, что Windows поступает с ней совсем не так, как с «родным» консольным приложением. Windows *эмулирует* исполнение DOS-программ, то есть пытается своими средствами выполнить программу так, чтобы никто не заметил подмены. Часто это удается. На моем компьютере с операционной системой Windows 98 до сих пор работает старинная электронная таблица LOTUS 1-2-3 v.2.2, написанная еще в 1989 году для системы DOS!

Как же Windows распознает программы для DOS? Так же как и для Windows — по самой программе, вернее, по ее заголовку. Ведь программа, хранящаяся в файле с расширением .exe, содержит не только инструкции процессора, но и сведения, которые требуются операционной системе для ее запуска.

Значит, программе, предназначенной для системы DOS, требуется особый заголовок, не такой, как у консольного приложения Windows. Такой заголовок созда-

ется специальным компоновщиком, который в нашей учебной версии ассемблера называется link16.exe[1]. Чтобы приготовить с его помощью программу для DOS, нужен специальный командный файл, показанный в листинге 8.2.

Листинг 8.2. Командный файл dmake.bat для создания DOS-программ

```
ml /c %1.asm
link16 %1.obj,%1.exe,,,,
```

Как видим, DOS-программа приготовляется тем же ассемблером, но другим компоновщиком. Запятые в командной строке, запускающей link16.exe, обозначают отсутствующие служебные файлы, которые нам не интересны.

Файл dmake.bat удобно поместить в ту же папку, что и amake.bat, создающий консольные приложения. Если сохранить программу из листинга 8.1 в файле l81.asm, то вызов командного файла с параметром l81

```
dmake l81
```

создаст программу l81.exe, которая запускается из командной строки FAR так же, как и консольное приложение Windows, и так же выводит на экран строку Не могу молчать!. Но мы-то знаем, что это другая программа, которую Windows исполняет совсем иначе.

Сегменты

Пожалуй, самое важное отличие программы, написанной для DOS, от консольного приложения Windows — в способах обращения с памятью. Строка

```
mov  dx, offset hello
```

[1] Этот компоновщик тоже написан для системы DOS, но отлично чувствует себя в среде Windows.

из листинга 8.1 кажется нам знакомой: по-видимому, в ней адрес начала последовательности символов Не могу молчать! записывается в регистр dx.

Но ведь dx — 16-битовый регистр и может содержать всего 2^{16} = 65536 различных адресов, что очень мало даже для такой старой системы как DOS. Почему же DOS не использует 32-битные регистры? Потому, что в процессорах — современниках DOS их просто не было! Процессор Intel 8086 — сердце компьютера IBM PC-XT — содержал только 16-битовые регистры ax, bx, cx, dx, si, di, и перед его разработчиками встал выбор: либо обречь процессор на работу с 65535 байтами, либо записывать адрес в двух регистрах.

Естественно, был выбран второй вариант. Решили организовать память в виде *сегментов*, каждый из которых содержит 64 килобайта или 64 Кбайт (64K = 64 * 1024 = 65535 байтов памяти). При этом положение байта внутри сегмента определяется обычным регистром, вроде bx, а положение самого сегмента внутри компьютерной памяти задается специальным *сегментным регистром*, каких в процессоре 8086 четыре: cs, ds, es, ss. Регистр cs задает сегмент, в котором находятся инструкции программы, регистр ss — положение стека, а регистры ds и es определяют положение сегментов данных. Поэтому обращение к памяти должно в общем случае содержать как смещение, так и сегментный регистр, например, инструкция

```
mov al, ds:[si]
```

пересылает байт, чей адрес складывается из адреса начала сегмента, хранящегося в регистре ds, и относительного адреса внутри сегмента, записанного в si. Правило, по которому определяется адрес начала сегмента, очень простое: нужно умножить содержимое сегментного регистра на 16. Но мы уже знаем, что умножение на 16 эквивалентно сдвигу числа на 4 двоичных разряда

влево. Выходит, максимальный адрес сегмента занимает всего 16 + 4 = 20 бит и равен $ffff0_{16}$ или 1048560_{10}. Если теперь к этому адресу прибавить 65535 — максимальное положительное число, способное уместиться в 16-битовом регистре), то получим максимальный адрес, который можно задать с помощью сегмента и смещения: чуть больше одного миллиона байтов!

Сейчас эта цифра кажется смехотворной, но когда процессор 8086 только появился, 1 мегабайт (миллион байтов) памяти был огромным числом, и разработчикам казалось, что программам его хватит на долгие годы.

Но уже через пару лет стало ясно, что они жестоко ошиблись. Электронная промышленность стала производить дешевые микросхемы памяти, только вот пользоваться ими было невозможно из-за предела в 1 Мбайт. Поэтому был разработан новый процессор 80286, в котором применялся другой способ адресации, позволявший использовать до 16 Мбайт памяти. Но чтобы на нем можно было выполнять старые программы, способные с помощью пары регистров сегмент-смещение адресовать только 1 Мбайт, пришлось в новом процессоре реализовать и старую систему адресации. Так возникли два режима процессора: *реальный* режим, совместимый с процессором 8086 и способный адресовать до 1 Мбайт памяти, и *защищенный* режим, устроенный совершенно иначе и способный адресовать до 16 Мбайт.

Это разделение на реальный и защищенный режимы сохранилось до сих пор во всех процессорах Intel. Начиная с процессора 80386, защищенный режим способен адресовать 2^{32} — более 4 миллиардов байтов! И опять это число, несколько лет назад казавшееся фантастическим, становится привычным, а для некоторых задач и недостаточным. Но о том, как преодолеть очередной барьер, мы в этой книге говорить не будем.

Вместо этого попробуем понять, как процессор взаимодействует с компьютерной памятью.

Мы уже говорили, что единичные и нулевые биты существуют только в головах программистов. Для процессора реальны только напряжения на его контактах. Каждый контакт соответствует одному биту, и процессору нужно различать только две градации напряжения: есть–нет, высокое–низкое. Одной из них соответствует единица, другой — ноль. Поэтому адрес для процессора — это последовательность напряжений на специальных контактах, называемых *шиной адреса*. Поскольку в реальном режиме процессора адрес состоит из 20 битов, в шине адреса процессора 8086 всего 20 контактов. Кроме контактов, на которых появляется адрес, в процессоре есть еще контакты, называемые *шиной данных*, где появляется прочитанное из памяти число. Шина данных процессоров 8086 и 80286 имеет 16 контактов, шина данных процессора 80386 и выше — 32 контакта.

Можно представить себе, что после того, как на контактах шины адреса, которыми кодируется двоичное число, *выставляются* напряжения, на контактах шины данных *появляются* напряжения, кодирующие хранящееся по указанному адресу число. Эта картина очень грубая, потому что для извлечения данных из памяти необходимо время. Чтобы не запутаться, работой процессора управляет специальный тактовый генератор. Он вырабатывает импульсы, которые делят работу процессора на отдельные шажки. Единицей времени процессора служит один *такт*, то есть промежуток между двумя сигналами тактового генератора (тактовыми импульсами). Некоторые команды выполняются за один такт. Обращение к памяти требует, как правило, нескольких тактов процессора. Когда данные поступили из памяти, на одном из контактов процессора появляется сигнал, говорящий о том, что данные готовы и их можно использовать в текущей инструкции.

Напряжения, появляющиеся на шине адреса процессора, называются *физическим* адресом. В реальном режиме процессор работает только с физическими адресами. Поэтому по сегменту и смещению всегда можно сказать, какие напряжения будут на 20 контактах адресной шины. Наоборот, защищенный режим процессора интересен тем, что программа работает с *логическими* адресами, а процессор незримо преобразует их в физические.

Наверное, вы уже догадались, что система Windows использует защищенный режим работы процессора. Современные операционные системы и программы требуют столько памяти, что защищенный режим работы процессора стал гораздо «реальнее» его реального режима. А это значит, что программы, написанные для DOS, тоже выполняются в защищенном режиме, то есть адреса, бывшие некогда физическими, такими быть перестали. Программе для DOS, операционная система выделяет *логическое* адресное пространство, которое не отличается от того, что было в реальном режиме. Но на самом деле система незаметно использует совсем другие адреса. Поскольку Windows — система многозадачная, она может выполнять одновременно множество программ для DOS, причем каждая DOS-программа чувствует себя так, как будто она одна выполняется процессором.

Опять про сегменты

Поскольку смещения в защищенном режиме процессоров 80386[1] и выше — 32-разрядные, программа для

[1] Системы Windows (Windows 95, 98, ME, 2000, XP) не могут работать с процессором 80286.

Windows использует, по существу, один огромный сегмент, занимающий 4 гигабайта (4 294 967 296 байтов) логического пространства. Раз сегмент один, его «настройку» берет на себя операционная система.

А в программе для DOS чаще всего приходится задавать несколько сегментов, потому что инструкции процессора и данные не всегда удастся уместить в 64 килобайта. Сегментов данных, как и сегментов кода, может быть много, поэтому в программе для DOS нужно явно «настроить» сегментные регистры. В нашей программе из листинга 8.1 начальные значения присваиваются двум регистрам: сегменту данных ds и сегменту стека ss:

```
mov  dx, @stack
mov  ss, dx
mov  dx, @data
mov  ds, dx
```

Имена @data и @stack обозначают значение регистра, которое станет известно в момент запуска программы. Ведь программа для DOS размещается по реальным, физическим адресам, поэтому значения сегментов заранее не известны и зависят от того, сколько памяти уже израсходовано операционной системой и другими, ранее запущенными программами, такими как резидентные программы и файловые оболочки, вроде Norton Commander. Процессор устроен так, что эти значения он не может непосредственно передать в сегментный регистр, приходится делать это через посредника (в нашем случае это регистр dx).

Мы уже говорили, что сегменты в DOS-программе очень невелики, и только одного сегмента данных .data, как в программе из листинга 8.1, может не хватить. Задать дополнительные сегменты можно с помощью директив .data? (см. раздел «Деление» главы 4) или .const. Последняя директива задает сегмент, хранящий всякие

постоянные величины: сообщения программы, константы с плавающей точкой и пр. Но лучше использовать в программах для DOS «классический» способ задания сегментов с помощью директивы segment. В листинге 8.3 показана программа, складывающая два числа, расположенных в разных сегментах данных data и data1.

Листинг 8.3. Сложение двух чисел, расположенных в разных сегментах

```
.8086
stack segment stack
db 100 dup (?)
stack ends
data   segment
first  dw 2
data   ends
data1  segment
second dw 3
data1  ends
code segment
assume cs:code, ds:data, es:data1, ss:stack
start:
mov  ax, data
mov  ds, ax
mov  ax, data1
mov  es, ax
mov  dx, first
add  dx, second
mov  ah, 4ch
int  21h
code ends
end  start
```

В этой программе задаются 4 сегмента. Строки

```
stack segment stack
db 100 dup (?)
stack ends
```

выделяют 100 байтов для сегмента стека.

Следом за сегментом стека задаются два сегмента данных data и data1. В каждом из этих сегментов расположено по одному числу. Это, конечно, глупость, и мы спокойно могли бы обойтись в этой программе одним сегментом. Просто задание двух сегментов данных позволяет лучше понять настройку сегментных регистров и выбор программой сегмента по умолчанию.

Вслед за сегментами данных идет кодовый сегмент

```
code segment
assume cs:code, ds:data, es:data1, ss:stack
start:
...
code ends
end  start
```

с новой для нас директивой assume, которая указывает ассемблеру, с каким сегментом будет связан определенный сегментный регистр. В нашем примере с сегментом data связан регистр ds, а с сегментом data1 — регистр es. Такую связь необходимо задать, чтобы ассемблер знал, какой сегментный регистр указать в соответствующей инструкции.

Возьмем, например, инструкцию mov dx, first, пересылающую число first в регистр dx. Чтобы эта инструкция имела какой-то смысл, ассемблер должен знать, какой сегментный регистр «подпирает» сегмент data, где хранится число first. Ведь адрес числа состоит из двух частей: смещения и сегмента. Так вот, директива assume как раз и говорит ассемблеру, что сегмент data связан с регистром ds.

И точно так же директива assume указывает ассемблеру, что сегментом data1 ведает регистр es, поэтому в инструкции add dx, second будет указан именно es.

Чтобы сказанное стало немного понятней, попробуем рассмотреть нашу программу в окне отладчика К сожалению, OllyDbg, который мы до сих пор использовали, не работает с программами для DOS, поэтому приходится использовать древний отладчик AfdPro —

ровесник системы DOS. Он тоже включен в наш учебный ассемблер и вызывается в оболочке FAR командой afdpro <имя программы>. Программа из листинга 8.2 смотрится в окне AfdPro примерно так, как на рис. 8.1.

```
AX 0000   SI 0000   CS 2EA0   IP 0000     Stack +0 0000  Flags 3202
BX 0000   DI 0000   DS 2E87                     +2 0000
CX 00A7   BP 0000   ES 2E87   HS 2E87           +4 0000  OF DF IF SF ZF AF PF CF
DX 0000   SP 0064   SS 2E97   FS 2E87           +6 0000   0  0  1  0  0  0  0

 CMD >                                          1        0  1  2  3  4  5  6
                                                DS:0000  CD 20 00 A0 00 9A F0 FE
                                                DS:0008  1D F0 18 05 4B 1B 6D 01
0000 B89E2E        MOV    AX,2E9E               DS:0010  26 14 78 01 26 14 18 1
0003 8ED8          MOV    DS,AX                 DS:0018  01 01 01 00 02 FF FF FF
0005 B89F2E        MOV    AX,2E9F               DS:0020  FF FF FF FF FF FF FF FF
0008 8EC0          MOV    ES,AX                 DS:0028  FF FF FF FF 71 2E C0 11
000A 8B160000      MOV    DX,[0000]             DS:0030  28 16 14 00 18 00 87 2E
000E 2603160000    ADD    DX,ES:[0000]          DS:0038  FF FF FF FF 00 00 00 0
0013 B44C          MOV    AH,4C                 DS:0040  07 0A 00 00 00 00 00
0015 CD21          INT    21                    DS:0048  00 00 00 00 00 00 00

2         0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
DS:0000   CD 20 00 A0 00 9A F0 FE  1D F0 1B 05 4B 1B 6D 01   - .a.bE•  .E..K.
DS:0010   26 14 78 01 26 14 18 16  01 01 01 00 02 FF FF FF   &.x.&...
DS:0020   FF FF FF FF FF FF FF FF  FF FF FF FF 71 2E C0 11                q.
DS:0030   28 16 14 00 18 00 87 2E  FF FF FF FF 00 00 00 00   (.....3.
DS:0040   07 0A 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........  ........

1 Step  ProcStep  Retrieve 4 Help ON  5 BRK Menu 6   7 up  8 dn   le 1 ri
```

Рис. 8.1. Программа в окне отладчика AfdPro

Первые 4 строки, видные в окне отладчика

```
0000 B89E2E  MOV     AX.2E9E
0003 8ED8    MOV     DS.AX
0005 B89F2E  MOV     AX.2E9F
0008 8EC0    MOV     ES.AX
```

присваивают начальные значения сегментным регистрам. Следующая строка, очевидно, представляет инструкцию mov dx. first:

```
8B160000     MOV   DX. [0000]
```

Здесь 8B160000 — шестнадцатеричный код инструкции, а MOV DX. [0000] — ее символическое представление. Видно, что имя переменной first ассемблер превратил в ее адрес 0000. Вернее, нули — это только смещение относительно какого-то сегмента. Если сегмент не указан, то процессор считает, что это ds. И действительно, директива assume закрепила за сегментом data, где хранится число first, именно этот регистр.

А теперь посмотрим, как показывает отладчик следующую инструкцию add dx. second:

```
2603160000     ADD    DX. ES:[0000]
```

Здесь символическое представление инструкции уже явно включает регистр ES: ADD DX. ES:[0000], что согласуется с директивой assume для сегмента data1, хранящего число second. Эта инструкция велит процессору взять число, чье смещение относительно сегмента es равно нулю. Любопытно узнать, где в коде команды хранится информация о том, что смещение отсчитывается именно относительно es. Оказывается, в инструкции 2603160000 это так называемый *префикс*, первые две шестнадцатеричные цифры. В нашем случае на регистр es указывают цифры 26. Заметим, что ассемблер ставит префиксы только там, где это необходимо. В команде 8B160000 (MOV DX. [0000]) нет префикса 3e, предусмотренного для регистра ds, потому что ds задается директивой assume и используется по умолчанию.

Эти правила умолчания довольно просты: при косвенной адресации, когда смещение операнда хранится в регистре, ассемблер считает, что регистры bx, si, di содержат смещения относительно ds, а bp — смещения относительно регистра стека ss[1].

Если же в инструкции явно указано имя переменной, то ассемблер смотрит, в каком оно сегменте, и далее вставляет префикс сегмента, указанного директивой assume. Если таким сегментом оказывается ds, префикс не ставится, потому что процессор использует ds по умолчанию.

[1] В процессоре 8086 только эти регистры участвуют в косвенной адресации. В процессорах 80386 и выше можно для этой же цели использовать регистры eax, ebx, ecx, edx, esi, edi, ebp.

Листинг 8.4. Программа находит нужный сегмент

```
.8086
stack segment stack
db 100 dup (?)
stack ends
data   segment
first  dw 2
data   ends
data1  segment
second dw 3
data1  ends
code segment
assume cs:code, ds:data, es:data1, ss:stack
start:
mov  ax, data
mov  ds, ax
mov  ax, data1
mov  es, ax
mov  bx, 0
mov  dx, [bx]
mov  ah, 4ch
int  21h
code ends
end  start
```

В программе из листинга 8.4 инструкции

```
mov  bx, 0
mov  dx, [bx]
```

не содержат никакой информации о сегменте. В них видно только нулевое смещение, которое имеют как число first в сегменте data, так и число second в регистре data1. Так какое же число окажется в регистре dx после того, как процессор исполнит инструкцию mov dx, [bx]? Легко проверить с помощью отладчика, что это будет двойка. Ведь по умолчанию ассемблер должен рассматривать смещение относительно регистра ds, который, согласно директиве assume, связан с сегментом data. Но, несмотря на директиву assume, регистр dx не «получит двойку», если явно не настроить сегмент ds инструкциями


```
mov  ax, data
mov  ds, ax
```

Обратите внимание: в программах из листингов 8.3, 8.4 начальные значения присваиваются только сегментным регистрам ds и es. Сегмент стека оставлен в покое, и это не случайность. Дело в том, что в объявлении сегмента стека

```
stack segment stack
...
stack ends
```

первое слово stack в строке stack segment stack может быть каким угодно, это просто название сегмента. А вот второе слово stack — служебное, оно показывает ассемблеру, что регистр стека ss надо настроить именно на этот сегмент. Поэтому в нашей программе нет явного присваивания значения сегменту ss, ведь это уже сделали за нас ассемблер и операционная система.

В заключение скажем несколько слов об отладчике AfdPro, заменяющем OllyDbg при работе с программами для DOS. Написанный в 80-х годах прошлого века немецким программистом Путкаммером (H.-P. Puttkammer), AfdPro неплохо смотрится и двадцать лет спустя.

AfdPro управляется командами, вводимыми с клавиатуры. Место, куда вводятся команды, помечено в окне отладчика значками CMD > (см. рис. 8.1). Самая важная команда отладчика — QUIT (выход). Набрав ее и нажав Enter, мы покидаем отладчик и видим уже синие панели оболочки FAR.

Самые важные клавиши, используемые при отладке программы, — F2 и F1. Первая выполняет программу по шагам, причем вызов и возврат из процедуры считается одним шагом. Клавиша F1 похожа на F2, но с ее помощью можно попасть внутрь процедуры и посмотреть, как выполняется каждая ее инструкция.

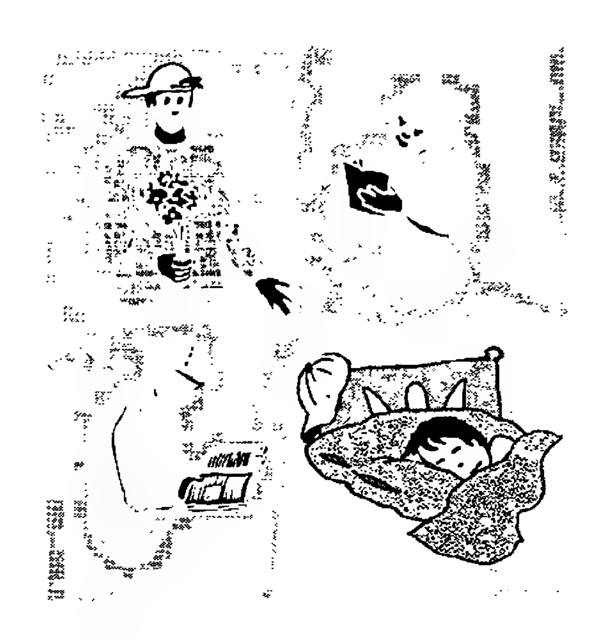
Регистры процессора и компьютерную память AfdPro показывает в нескольких окнах. Вверху видны регистры и флаги, внизу — память (шестнадцатеричные коды и соответствующие им символы). В окне справа показана та же память, но без символьного представления.

«Забираться» в различные окна отладчика позволяют клавиши F7, F8 (движение вверх-вниз) и F9, F10 (вправо-влево). Попав в окно, позволяющее увидеть память компьютера, можно изменить не только сегментный регистр, но и любой байт. Естественно, память можно просматривать в любом направлении с помощью клавиш ↓↑.

Результат работы программы можно увидеть, переключаясь между окном отладчика и экраном компьютера с помощью клавиши F6. Но прежде необходимо выполнить команду mo a on (показана на рис. 8.1).

В отладчике AfdPro очень много возможностей, полное описание которых потребовало бы целой книги — никак не меньше той, что вы держите сейчас в руках. Но AfdPro понятен и так, а большая часть его команд описана в файле помощи, вызываемом клавишей F4.

ГЛАВА 9 Жизнь в сегментах



Ужимки и прыжки

Нас посылают куда подальше. Благодаря этому мы движемся.

Аркадий Давидович. «Афоризмы»

Сегменты разобщают компьютерную память, ставят в ней множество ненужных перегородок — сегментов, похожих на маленькие темные клетушки в изначально просторном и светлом офисе. Подобно служащему, вынужденному открывать множество дверей, путешествуя от одной комнатки к другой, программист должен каждый раз думать над тем, куда и как переходит процессор, чтобы указать ему кратчайший путь. Правильно указанный переход не только «ужимает» (делает короче) программу, но и заставляет ее быстрее выполняться.

Самый простой и близкий переход позволяет отправить процессор на 128 байт назад или на 127 — вперед. Эти числа возникли не случайно, потому что длина прыжка кодируется в самой инструкции и занимает один байт, способный хранить числа от −128 до 127. Всего такая инструкция перехода занимает два байта. В следующем фрагменте программы

```
mov ax, 2    ;0000 B80200  MOV  AX,0002
jmp exit     ;0003 EB03    JMP  0008
mov ax, 3    ;0005 B80300  MOV  AX, 0003
exit:        ;0008
```

показаны инструкции ассемблера и (в комментариях) соответствующие им коды и адреса, видные в окне отладчика. Так, например, инструкция `mov ax, 2` имеет

смещение 0000 (относительно сегмента cs) и занимает три байта. Ее код b80200, очевидно, содержит признак операции (B8) и само прибавляемое число 0002, но только вывернутое наизнанку по законам процессора Intel.

Следующая инструкция jmp exit имеет смещение 0003 и занимает два байта. Первый из них (EB) определяет саму инструкцию (процессор понимает, что перед ним ближний переход в пределах 127 байт), а второй — длину прыжка относительно следующей инструкции. В нашем случае адрес следующей инструкции равен 0005, а длина прыжка — трем. Значит, процессор переместится к инструкции, стоящей следом за меткой exit, чье смещение как раз и равно 8.

В этом разделе мы, пожалуй, впервые обратили внимание на двоичные коды инструкций процессора. Чем опытнее программист, тем больше он смотрит на эти коды и тем меньше — на инструкции ассемблера. Настоящие мастера способны читать прямо .exe-файлы, даже не заглядывая в исходные тексты, но нам до этого далеко. Будем пока интересоваться кодированием инструкций перехода, что поможет нам понять, почему их создано так много.

Следующим по сложности переходом будет команда jmp, занимающая в памяти три байта и потому способная послать процессор на 32 768 байт назад и на 32 767 байт — вперед. Тот же самый отрывок программы, но уже с другим переходом, будет таким:

```
mov ax, 2           :0000 B80200 MOV AX, 0002
jmp near ptr exit   :0003 E90300 JMP 0009
mov ax, 3           :0006 B80300 MOV AX, 0003
exit:               :0009
```

Здесь, в отличие от предыдущего примера, инструкция перехода jmp занимает три байта, и код ее начинается уже байтом E9, а не EB, как в прошлый раз. По это-

му байту процессор поймет, что перед ним инструкция перехода, занимающая три байта, и будет рассматривать следующие два байта как длину прыжка относительно начала следующей инструкции, равную в нашем случае трем. Обратите внимание на то, как изменился текст программы. Вместо простого jmp exit стоит jmp near ptr exit. Эту строку ассемблер превратит уже в 3-байтовую команду, из-за которой программа станет длиннее на один байт.

Следующий переход предназначен для путешествия «куда подальше» — в другой сегмент, и будет полезно познакомиться с ним на примере программы, показанной в листинге 9.1.

Листинг 9.1. Путешествие в другой сегмент

```
.8086
stack segment stack
db 100 dup (?)
stack ends
code1  segment
assume cs:code1
addd:
mov    ax, 2
add    ax, 3
jmp far ptr exit
code1  ends
code segment
assume cs:code, ss:stack
start:
jmp  far ptr addd
exit:
mov  ah, 4ch
int  21h
code ends
end  start
```

В ней заданы два кодовых сегмента — code и code1. Переход в другой сегмент задается инструкцией jmp far ptr addd, затем в сегменте code1 складываются два числа, после чего инструкция jmp far ptr exit возвращает процессор в сегмент code. Инструкция jmp far ptr addd выглядит в окне отладчика так:

```
EA00009E2E     JMP    2E9E:0000
```

Она, как видите, занимает уже 5 байт памяти и содержит *абсолютный* адрес, состоящий из сегмента 2E9E и смещения 0000.

Программа, показанная в листинге 9.1, — одна из самых глупых в этой книге. Чтобы сложить два числа, незачем тащиться в чужой сегмент. Но она служит хорошим пособием по дальним переходам, а большего нам и не надо.

Подумаем, например, над тем, всегда ли нужно указывать ассемблеру, что предстоит дальний переход. Очевидно, строка

```
jmp far ptr exit
```

необходима, потому что ассемблер, встретив ее, еще не знает, что метка exit находится в другом сегменте. Ведь ассемблер MASM — *однопроходный*, то есть читает текст программы только раз — сверху вниз. А вот второй переход

```
jmp  far ptr addd,
```

вроде бы не нуждается в операторе far ptr, ведь ассемблер, когда встретит команду перехода, уже знает, что addd — «чужая» метка, расположенная в другом сегменте. Но ассемблер откажется компилировать программу, в которой переход записан как jmp addd. Все равно придется явно указать ему, что addd — дальняя метка, написав addd label far вместо addd:, и только тогда, программа, чей отрывок показан ниже, станет работать.

```
code1  segment
assume cs:code1
addd label far
...
code1  ends
```

```
code segment
assume cs:code, ss:stack
start:
jmp addd
...
```

Помимо прямых переходов разной дальности, с которыми мы только что познакомились, есть еще и *косвенные* переходы по адресу, задаваемому в регистре или памяти компьютера:

```
mov ax, 2      ;0000 B80200 MOV AX, 0002
mov dx, offset exit;0003 BA0B00 MOV DX,000B
jmp dx         ;0006 FFE2   JMP  DX
mov ax, 3      ;0008 B80300 MOV AX, 0003
exit:          ;000B
```

Здесь адрес перехода посылается сначала в регистр dx инструкцией mov dx,offset exit, а затем уже происходит переход по указанному в этом регистре адресу jmp dx. Обратите внимание: этот адрес абсолютный, а не относительный, как в предыдущих примерах.

Естественно, косвенный переход может быть не только ближним. Чтобы перескочить в другой сегмент, нужно записать в двойное слово памяти значение этого сегмента и смещение — примерно так, как в программе из листинга 9.2.

Листинг 9.2. Косвенный переход в другой сегмент

```
.8086
$w equ word ptr
$o equ offset
stack segment stack
db 100 dup (?)
stack ends
code1  segment
assume cs:code1
addd:
mov    ax, 2
add    ax, 3
jmp far ptr disp ;возврат
code1  ends
code segment
assume cs:code, ss:stack
start:
mov $w faddr, $o addd
mov $w faddr[2], SEG addd
jmp faddr ;дальний переход
disp:
mov  ah, 4ch
int  21h
faddr dd ?
code ends
end  start
```


В этой программе несколько новшеств. Во-первых, двойное слово faddr, хранящее адрес дальнего перехода, расположено в кодовом сегменте, на что имеет полное право. Инструкции и данные могут находиться рядом, если процессор сможет отличить одно от другого. В нашем случае их нельзя спутать, потому что faddr находится «в тени» — после инструкций завершения программы.

Далее, в программе применяется сокращенная запись операторов (вместо word ptr пишем $w, а вместо offset — просто $o). Эти сокращения определены в самом начале программы директивами equ.

Наконец, сама организация дальнего перехода, чей адрес (по обычаю процессоров Intel) хранится в двойном слове faddr, вывернутым наизнанку: сначала смещение (как младшая часть адреса), потом сегмент. Для вычисления сегментного адреса метки в ассемблере есть специальный оператор SEG. Адрес сегмента записывается во вторую половину двойного слова faddr инструкцией

```
mov $w faddr[2], SEG addd,
```

после чего сам переход оказывается крайне простым:

```
jmp faddr
```

Межсегментные каналы

> Волго-Дон уникален тем, что соединил моря севера и юга, в два раза уменьшив водное расстояние между ними. Поэтому Волго-Донской канал без преувеличения — сокровище России.
>
> *Журнал «Деловые вести»*

Некоторые инструкции процессора нарочно созданы для того, чтобы преодолеть разобщенность сегментов и построить между ними подобие канала. Такова группа инструкций movs, позволяющих передать байт (movsb), слово (movsw) и двойное слово (movsd)[1] из одного сегмента в другой.

Чтобы «соединить моря севера и юга», инструкцию movs нужно настроить так, чтобы пара сегментов ds:si содержала адрес переменной-источника, а es:di — адрес переменной-приемника. После чего содержимое переменной с адресом ds:si будет скопировано инструкцией movs в новое место по адресу es:di. Если при этом флаг направления D опущен, то si и di синхронно увеличатся на число копируемых байтов (в нашем случае на 2). И если повторно выполнить инструкцию movsw, скопируется следующее слово. Программа из листинга 9.3 переписывает слово из сегмента north_sea в сегмент south_sea.

Листинг 9.3. Переписывание слова из одного сегмента в другой

```
.286
stack segment stack
db 100 dup (?)
stack ends
north_sea   segment
src dw  3
north_sea  ends
```

[1] Двойные слова под силу только процессору 80386 и выше.


```
south_sea   segment
dst dw ?
south_sea  ends
code segment
assume cs:code, ds:north_sea, es:south_sea assume
ss:stack
start:
mov  ax, north_sea
mov  ds, ax
mov  ax, south_sea
mov  es, ax
mov  si, offset src
mov  di, offset dst
movsw
mov  ah, 4ch
int  21h
code ends
end  start
```

Канал, устроенный инструкцией movs, не кажется очень эффективным — слишком много нужно приготовлений для пересылки одного слова. Но вспомним о префиксе rep, с которым познакомились в разделе «Командная строка» главы 6. С помощью rep инструкция movs может передать из одного сегмента в другой сколько угодно слов, что оправдает хлопоты, связанные с ее настройкой. Фрагмент программы, пересылающей 100 слов из одного сегмента в другой, может быть таким:

```
mov  ax, north_sea
mov  ds, ax
mov  ax, south_sea
mov  es, ax
mov  si, offset src
mov  di, offset dst
mov  cx, 100
cld
rep movsw
```

Обратите внимание на инструкцию cld, которая опускает флаг направления, задавая тем самым авто-

матическое *увеличение* адресов при передаче данных между сегментами.

Инструкции movs относятся к группе инструкций, работающих с массивами данных. С одной такой инструкцией scas, сравнивающей байт (слово, двойное слово) с адресом es:di и байт (слово, двойное слово) в регистре al (ax, eax), мы уже познакомились в разделе «Командная строка» главы 6[1]. Будет логично упомянуть в этом разделе и другие полезные инструкции из этой группы.

Всего ближе к movs пара инструкций lods, stos. Первая читает байт (слово, двойное слово) по адресу ds:si и записывает его в регистр al (ax, eax). Вторая читает байт (слово, двойное слово) из регистра al (ax, eax) и записывает его по адресу es:di. Обе инструкции увеличивают или уменьшают (в зависимости от флага направления D) регистры si (di) на число прочитанных (переданных) байтов.

Из пары инструкций lods, stos можно составить инструкцию movs:

```
mov  ax, north_sea
mov  ds, ax
mov  ax, south_sea
mov  es, ax
mov  si, offset src
mov  di, offset dst
lodsw   ;прочитать слово
;<здесь сообщение можно перехватить>
stosw   ;записать слово
```

Как видите, между lodsw и stosw можно поставить подслушивающее устройство, способное запоминать и менять передаваемые данные.

[1] Там инструкция scasb использовалась в консольном приложении Windows и потому не нуждалась в установке сегментных регистров ds и es.

Кроме упомянутых есть еще инструкция cmps, которая не передает данные между сегментами, а сравнивает их между собой. Такая инструкция полезна, когда нужно найти отличия во внешне похожих массивах данных.

Попробуем, например, сравнить два почти одинаковых «стога», которые отличаются тем, что в одном есть иголка, а во втором — нет. Первый стог хранится в сегменте hay1, второй — в сегменте hay2 (листинг 9.4).

Листинг 9.4. Поиск «иголки» в стоге «сена»

```
.8086
stack segment stack
db 100 dup (?)
stack ends
hay1   segment
equal  db "Равны", 13, 10, '$'
nequal db "Не равны", 13, 10, '$'
src db "сеносеносеносеносеносено"
zsize   dw ($-src)
hay1  ends
hay2   segment
dst db  "сеносеноиголкасеносеносе"
hay2  ends
code segment
assume cs:code, ds:hay1, es:hay2, ss:stack
start:
cld
mov  ax, hay1
mov  ds, ax
mov  ax, hay2
mov  es, ax
mov  si, offset src
mov  di, offset dst
mov  cx, zsize
repe cmpsb
mov  dx, offset nequal
cmp  cx, 0
jnz  disp
mov  dx, offset equal
```

продолжение ⇗

Листинг 9.4 *(продолжение)*

```
disp:
mov  ah, 09
int  21h
mov  ah, 4ch
int  21h
code ends
end  start
```

Программа, показанная в листинге, сравнивает две последовательности символов. Первая находится в сегменте hay1 и помечена как src, вторая (с меткой dst) хранится в сегменте hay2. В центре этой довольно длинной программы инструкция repe cmpsb, сравнивающая последовательности символов. Инструкция cmps, подобно movs, после каждого сравнения увеличивает (или уменьшает, если поднят флаг направления) si и di на число сравниваемых за раз байтов (в нашем случае на 1).

Чтобы инструкция cmpsb работала правильно, ее нужно подготовить так же, как инструкцию movs: задать сегментные регистры и смещения строк. В регистр cx посылается размер сравниваемых строк zsize, вычисляемый ассемблером в процессе компиляции. Мы уже встречались с таким способом в разделе «Переходы» главы 4. Префикс repe означает «повторять пока равно». Если строки идентичны, то процессор сделает столько сравнений, сколько указано в регистре cx. В этом случае cx будет равен нулю после выполнения всех инструкций repe cmpsb. Если же строки отличаются, инструкции cmps прекратят выполняться и cx будет отличен от нуля. В зависимости от того, равен или не равен cx нулю, программа покажет на экране фразу Равны или Не равны. Заметим, что сообщения Равны, Не равны должны быть именно в сегменте hay1, потому что процедура DOS, показывающая их на экране, требует, чтобы смещение сообщения, засылаемое в регистр dx инструкцией


```
mov  dx, offset <сообщение>
```

было относительно сегмента ds.

Инструкции scas, movs, cmps, lods, stos, с которыми мы только что познакомились, работают и в консольных приложениях Windows. Но там у программы всего один сегмент, поэтому инструкциям нужны только смещения, что сильно упрощает работу не только с этими, но и со всеми остальными инструкциями.

Нам осталось сказать, что префиксы rep, repe, repne (повторять пока не равно) работают только с инструкциями scas, movs, cmps, stos. Бессмысленно использовать их с такими инструкциями как mov.

Процедуры

Созданная в разделе «Ужимки и прыжки» программа (листинг 9.1) демонстрирует дальний переход в чужой сегмент, где складываются два числа, и дальний же возврат в основную программу. То, что она проделывает, больше всего напоминает вызов процедуры, которая может вернуться только к метке exit в основной программе. Так, конечно, делать, нельзя: необходимо превратить инструкции в процедуру, которая возвращается, подобно бумерангу, точно в то место, откуда была запущена.

Мы уже хорошо знаем, что все это делается с помощью инструкций call и ret. Правда, в случае DOS приходится думать, какой процедуре нужен вызов (далекий или близкий) и какой возврат. Программа, показанная в листинге 9.5, вызывает дальнюю процедуру, расположенную в «чужом» сегменте code1.

Листинг 9.5. Дальний вызов процедуры

```
.8086
stack segment stack
db 100 dup (?)
stack ends
code1  segment
assume cs:code1
f_add proc far
mov    ax, 2
add    ax, 3
ret         ;CB RET Far
f_add   endp
code1  ends
code segment
assume cs:code, ss:stack
start:
call f_add  :9A00009E2E CALL   2E9E:0000
mov  ah, 4ch
int  21h
code ends
end  start
```

Процедура f_add объявлена как f_add proc far. Это значит, что ей нужен дальний вызов с указанием сегмента и смещения и дальний же возврат. То есть инструкция ret в процедуре должна доставать из стека сегмент и смещение, предварительно сохраненные там еще до ее вызова.

Что касается вызова процедуры, то он будет по умолчанию дальним, раз она находится в другом сегменте. А вот возврат получился дальним из-за того, что процедура объявлена как far.

В листинге 9.5 инструкции вызова процедуры и возврата показаны в комментариях такими, какими видит их отладчик. В инструкции вызова call явно указаны сегмент и смещение:

```
9A00009E2E CALL   2E9E:0000
```

а вместо инструкции ret отладчик показывает дальний возврат ret far: CB RET Far, который достает из стека два

слова: сначала смещение, а затем сегмент. Получается так потому, что при вызове процедуры последним сохраняется смещение, ведь стек растет в сторону уменьшения адресов, и, согласно правилам процессора Intel, младшая часть двойного слова (смещение) должна иметь меньший адрес.

Очевидно, ассемблер ставит инструкцию дальнего возврата, потому что процедура объявлена дальней (far). Не будь этого словечка, процедура считалась бы по умолчанию ближней, и код инструкции возврата был бы уже другим (C3). Нужную инструкцию возврата можно задать и вручную: дальний возврат записывается как retf, а ближний — retn.

Вручную можно выполнить и вызов процедуры. Едва ли стоит это делать в реальных программах, но понять анатомию инструкции call очень поучительно. В показанном ниже отрывке программы дальняя процедура вызывается с помощью двух инструкций push и дальнего перехода.

```
start:
push cs
mov  ax, offset exit
push ax
jmp far ptr f_add
exit:
…
code ends
end  start
```

Эти два заталкивания в стек и следующий за ними дальний переход очень напоминают инструкцию call, только иначе записанную. Перейдя к началу процедуры и выполнив все, что требуется, процессор встретит на выходе инструкцию дальнего возврата, которая направит его туда, куда указывают сохраненные смещение и сегмент.

Заметим, что сохранение адреса возврата в стеке с последующим дальним переходом отличается от инструкции call тем, что затолкнуть в стек можно любой адрес, а не только адрес инструкции, непосредственно следующей за вызовом call. То есть сочетая сохранения в стеке и дальний переход jmp far, можно заставить процедуру возвратиться (с помощью ret) куда угодно.

Инструкцию retf можно использовать и вне процедуры, чтобы выполнить замаскированный дальний переход. Для этого нужно перед retf сохранить в стеке нужный адрес. В программе из листинга 9.6 с помощью инструкции retf как раз и совершается переход к метке target, находящейся в другом сегменте.

Листинг 9.6. Замаскированный переход к метке target

```
.8086
stack segment stack
db 100 dup (?)
stack ends
code1  segment
assume cs:code1
target:
jmp far ptr exit
code1  ends
code segment
assume cs:code, ss:stack
start:
mov ax, SEG target
push ax
mov ax, offset target
push ax
retf
exit:
mov  ah, 4ch
int  21h
code ends
end  start
```


Сначала в стеке сохраняется сегментный адрес метки

```
mov ax, SEG target
push ax
```

Затем ее смещение

```
mov ax, offset target
push ax
```

А сам переход выполняет инструкция дальнего возврата retf. Аналогично выполняется и ближний переход. Нужно только использовать retn вместо retf и сохранить в стеке одно смещение.

До сих пор мы вызывали процедуру, расположенную в другом сегменте. Когда же она находится в «родном» сегменте, все упрощается. Если процедура должна вызываться извне и потому объявлена как far, можно использовать дальний вызов call far ptr <имя>. Если же вызывать такую процедуру как ближнюю инструкцией call <имя>, то ассемблер автоматически вставит перед вызовом инструкцию push cs, чтобы правильно сработал дальний возврат. Так поведет себя ассемблер Masm. В сомнительных случаях программу нужно обязательно проверять отладчиком и вручную вставлять инструкцию push cs, если ассемблер этого не делает сам.

В частности, push cs приходится вставлять вручную при косвенном вызове подпрограммы, показанном в листинге 9.7.

Листинг 9.7. Косвенный вызов подпрограммы

```
.8086
stack segment stack
db 100 dup (?)
stack ends
code segment
assume cs:code, ss:stack
start:
```

продолжение ⇗

Листинг 9.7 *(продолжение)*

```
mov bx. offset f_add
push cs      ;сохранить сегмент
call bx      ;вызов f_add
mov  ah. 4ch
int  21h
f_add proc far
mov   ax. 2
add   ax. 3
ret
f_add endp
code ends
end  start
```

Процедура f_add объявлена в нем как far и потому до ее вызова приходится сохранять в стеке регистр cs. Инструкция call bx осуществляет ближний вызов процедуры, то есть сохраняет в стеке регистр bx, хранящий смещение f_add, и потом переходит к самой метке f_add. Но перед вызовом в стеке был сохранен еще сегментный регистр, что обеспечит правильный дальний возврат.

Завершим этот раздел примерами косвенного вызова процедуры, когда ее адрес хранится в памяти компьютера, а не в регистре (листинг 9.8).

Листинг 9.8. Вызов процедуры. чей адрес хранится в памяти

```
.8086
stack segment stack
db 100 dup (?)
stack ends
code segment
assume cs:code. ss:stack
start:
push cs
call nearp
call farp
mov  ah. 4ch
int  21h
f_add proc far
mov   ax. 2
```



```
add   ax. 3
ret
f_add endp
nearp dw f_add
farp  dd f_add
code ends
end  start
```

В нем сначала совершается ближний вызов call nearp, где nearp — слово в компьютерной памяти, хранящее смещение процедуры. Поскольку сама процедура — дальняя, перед ее вызовом в стек загружается cs. Второй вызов процедуры call farp ассемблер автоматически делает дальним, потому что farp — двойное слово, содержащее (как надеется ассемблер) сегмент и смещение.

Без всякого сомнения, самое сложное в этом примере — объявления переменных nearp и farp:

```
nearp dw f_add
farp  dd f_add
```

Мы привыкли, что метка — это *содержимое* переменной. В этом нас, казалось бы, убеждает отрывок программы

```
mov ax.digit:0000 2EA10800 MOV AX.CS:[0008]
...
digit dw 3  :0008 0300.
```

после исполнения которого в регистре ax оказывается тройка. Но если посмотреть код инструкции mov ax. digit (показан в комментарии), то окажется, что ассемблер превращает метку digit в *адрес* числа 3. Так что метка для ассемблера — это адрес. И вместо mov ax. digit разумнее писать mov ax. [digit], как бы говоря себе о том, что в регистр ax посылается содержимое слова с *адресом* digit. Вот почему переменная nearp в нашем примере хранит *адрес* метки nearp, а вовсе не содержимое памяти с такой меткой.

Адресация

Адреса содержатся и во всех других инструкциях ассемблера, имеющих дело с переменными, хранящимися в памяти. Нам уже знакома косвенная адресация mov ax. [bx][1]. где адрес (смещение) слова в памяти хранит регистр bx. Встречалась нам и адресация, полезная при работе с массивами

```
mov al. array[si].
```

где si добавляется к адресу начала массива array и в результате получается адрес его элемента под номером si[2]. Процессоры 8086 и 80286 могут использовать для такой адресации 4 регистра: bx, bp, si, di.

Мы уже привыкли к тому, что при адресации в квадратных скобках стоит нечто, содержащее адрес. А поскольку array и si в нашем последнем примере тоже образуют адрес, то можно заключить их в квадратные скобки и записать инструкцию так:

```
mov al. [array + si].
```

Конечно, это только другая запись; код инструкции, сгенерированный ассемблером, будет и в том и другом случае одинаковым.

Кроме перечисленных процессоры 8086 и 80286 могут использовать и другие способы адресации с использованием двух регистров. Разные варианты такой адресации показаны на рис. 9.1.

$$\left[\begin{Bmatrix} BX \\ BP \end{Bmatrix} + \begin{Bmatrix} SI \\ DI \end{Bmatrix} + \{\text{число}\}\right]$$

Рис. 9.1. Адрес может быть суммой двух регистров и смещения

[1] При косвенной адресации для процессоров 8086 и 80286 можно использовать только 3 регистра: bx, si, di.

[2] Только если в массиве хранятся байты.

Чтобы указать правильный адрес, нужно взять по регистру из каждой колонки и (если это необходимо) добавить смещение. Ассемблер согласится искать в памяти переменную с адресом [bx + si] или [bp + di][1], но окажется бессилен сделать что-нибудь с адресом [bx + + bp] или [si + di].

В программе из листинга 9.9 показано, как можно использовать новую адресацию для записи чисел в одномерный массив array.

Листинг 9.9. Адресация с помощью двух регистров

```
.8086
ARRSIZE equ 20
stack segment stack
db 100 dup (?)
stack ends
code segment
assume cs:code. ds:code. ss:stack
start:
mov  bx. offset array
mov  si. 5
shl  si. 1
mov  word ptr [bx+si]. 3
mov  ah. 4ch
int  21h
array dw ARRSIZE dup (?)
code ends
end  start
```

В ней адрес начала массива загоняется в регистр bx инструкцией mov bx. offset array. Далее в регистр si записывается число 5 — номер элемента массива. А поскольку в массиве array хранится ARRSIZE *слов*, то si нужно еще умножить на 2, чтобы получить *адрес* элемента относительно начала массива. А дальше инструкция mov word ptr [bx+si]. 3 записывает число 3 в пятый элемент массива.

[1] Если в адресе есть регистр bp, то по умолчанию адресация идет относительно сегмента стека ss.

Заметим, что адрес [bx+si] можно представить как [bx][si]. Для ассемблера обе записи эквивалентны и потому будут превращены в одну и ту же инструкцию процессора.

Как видите, способов адресации для процессоров 8086 и 80286 довольно много. Но с появлением процессора 80386 их стало настолько больше, что глядя на рис. 9.2, где они показаны, можно подумать, что речь идет совсем о другом процессоре.

$$\left[\begin{Bmatrix}EAX\\ECX\\EDX\\EBX\\ESP\\EBP\\ESI\\EDI\end{Bmatrix} + \begin{Bmatrix}EAX\\ECX\\EDX\\EBX\\EBP\\ESI\\EDI\end{Bmatrix} * \begin{Bmatrix}1\\2\\4\\8\end{Bmatrix} + \{\text{число}\}\right]$$

Рис. 9.2. Способы адресации для процессора 80386

Чтобы указать адрес для процессора 80386, достаточно заключить в квадратные скобки один из регистров из левой колонки [edx], или один из регистров из следующей колонки (умноженный на 2, 4, 8) [esi*2], или просто число [4856], или же число, но представленное меткой [label], или, наконец, любую комбинацию разных колонок (не обязательно всех), в которой регистры не совпадают, например:

```
[eax + edx*8 + 42]
```

Увидев в квадратных скобках эти регистры, ассемблер создаст инструкцию, которая сложит содержимое eax с числом, хранящимся в edx, умноженным на 8,

и прибавит к полученной сумме 42[1]. Полученное число будет для процессора адресом переменной, с которой ему придется сделать то, что приказано.

Всего чудеснее в такой адресации возможность умножать регистры, стоящие во второй колонке на 2, 4, или 8, что автоматически позволяет сформировать адрес нужного элемента массива, пользуясь регистром как индексом. Если переписать программу из листинга 9.9 для процессора 80386, то запись числа 3 в пятый элемент массива array выглядела бы так:

```
mov  ebx. offset array
mov  esi. 5 ;esi - индекс
mov  word ptr [ebx+esi*2]. 3 ;esi*2 = адрес
```

или еще проще:

```
mov  esi. 5
mov  word ptr array[esi*2]. 3
```

Число способов адресации кажется чрезмерным (особенно для процессора 80386), хотя наверняка найдутся задачи, где можно с пользой применить самые сложные из них. Но оказывается, адресацию можно использовать там, где нет и речи об адресе!

Ведь адрес — это всегда некое арифметическое выражение, где к регистру прибавляется другой регистр, умноженный на двойку, четверку или восьмерку, а к полученной сумме прибавляется (или из нее вычитается) произвольное число. Причем процессор вычисляет это выражение где-то в своих недрах, «разом», ведь результат должен использоваться как адрес. Но не обязан. Полученную сумму можно считать не адресом, а просто суммой чисел, которая вычисляется для чего-то другого.

Эту способность процессора легко вычислять арифметические выражения определенного вида использует

[1] Число можно и вычесть, то есть возможен и такой адрес: [eax + + edx*8 - 42].

инструкция lea, чье название состоит из первых букв английской фразы *Load Effective Address* (загрузить эффективный адрес). Чтобы понять, как она работает, сравним две инструкции:

```
mov  eax, [ebx+esi*2]
lea  eax, [ebx+esi*2]
```

Первая посылает в регистр eax содержимое двойного слова с адресом ebx + esi * 2. Вторая посылает в eax *сам адрес*, то есть сумму ebx и умноженного на 2 регистра esi. Эта сумма может быть адресом, а может и не быть, и потому мы вольны использовать ее как угодно.

Но поскольку изначальный смысл инструкции lea все-таки в получении адреса, ее можно использовать так же, как и оператор offset. Инструкции

```
mov  bx, offset arr ;BB1400    MOV BX, 0014
lea  bx, arr        ;8D1E1400  LEA BX, [0014]
```

посылают в регистр bx одно и то же число — адрес, связанный с меткой arr. Но инструкция lea занимает больше места в памяти, и потому оператор offset может быть выгодней там, где эту память приходится экономить.

Прерывания

В любой операционной системе есть набор стандартных процедур, с помощью которых программа взаимодействует с внешней для нее средой: клавиатурой, экраном монитора, музыкальной платой, сетевой картой, последовательным портом и самой операционной системой. Мы уже знакомы с некоторыми процедурами Windows API, такими как WriteConsole или ExitProcess. Они, как мы помним, вызываются так же, как и обычные процедуры ассемблера.

В системе DOS все устроено иначе. DOS API — это набор особенных процедур, называемых *прерываниями*. У каждого прерывания есть номер и параметры, которые передаются в регистрах процессора.

Так, например, прерывание INT 21h, с помощью которого на экран выводится строка символов, управляется двумя параметрами: в регистре ah должно быть число 9, а в регистре dx — адрес первого байта (относительно сегмента ds) строки символов, оканчивающейся значком $ (см. листинг 8.1).

Прерывания под номером 21h (33 – в десятичной системе счисления), чье действие определяется регистром ah, называются *функциями DOS*, у них нет названий, а только номера. Говоря о девятой функции DOS имеют в виду прерывание 21h с параметром ah, равным 9.

Различных функций DOS порядка сотни. Многие книги содержат их полное описание[1]. Но гораздо удобнее пользоваться компьютерными справочными системами вроде Norton Guide или списком прерываний Ральфа Брауна[2]. Поэтому мы, вместо того чтобы знакомиться с конкретными прерываниями, попробуем понять, как все они работают.

Прерывания, как мы уже поняли, — это разновидность процедур. Выполнив прерывание, процессор возвращается к следующей за ним инструкции — так же, как и после вызова процедуры. Но в отличие от процедуры перед вызовом прерывания процессор сохраняет в стеке текущей программы не только сегмент и смещение следующей команды, но и регистр флагов! Почему он так делает, мы поймем чуть позже, а пока ясно,

---

[1] Например, Р. Данкан. Профессиональная работа в MS-DOS, М.: Мир, 1993.

[2] И Norton Guide, и список прерываний легко найти в Интернете. Достаточно поискать в системе Google (www.google.com) фразы «Norton Guide» и «Ralf Brown's Interrupt List».

что обычная инструкция ret не годится для выхода из прерывания, потому что она достает из стека только два регистра, а поскольку регистр флагов сохраняется в стеке последним, инструкция ret достанет из стека совсем не то, и процессор безнадежно запутается. Поэтому для выхода из прерывания существует специальная инструкция iret (Interrupt Return — «Возврат из прерывания»), которая загоняет в регистр флагов содержимое вершины стека, затем достает из стека сегмент и смещение следующей за прерыванием команды и отправляет по этому адресу процессор. Заметим, что прерывания всегда дальние, то есть инструкция int <номер> сохраняет в стеке обязательно и сегмент и смещение следующей инструкции, а сам процессор тоже идет «куда подальше» — адрес перехода к прерыванию всегда состоит из сегмента и смещения.

Осталось понять, что это за адрес, то есть куда идет процессор, после того как инструкция прерывания сохранила в стеке адрес возврата и регистр флагов. Оказывается, адрес «куда пойти» содержится в специальной таблице, занимающей в компьютере, работающем под управлением DOS, первые 1024 байта памяти. Адрес нулевого прерывания хранится в первых 4 байтах этой таблицы (сначала смещение, затем сегмент). Адрес прерывания 21h занимает в этой таблице 33 место. Зная номер прерывания, процессор просто умножает его на 4, затем обращается к таблице и получает там адрес перехода. Увидеть этот адрес можно и вручную, если правильно настроить один из сегментных регистров. Например, адрес перехода для прерывания 21h можно получить так:

```
mov  ax, 0
mov  es, ax        ;es = 0
mov  bx, 21h       ;номер прерывания
shl  bx, 2         ;умножим на 4
mov  ax, es:[bx]   ;смещение
mov  dx, es:[bx+2] ;сегмент
```


Так определяются адреса перехода для прерываний в системе DOS. В Windows нет ни сегментов, ни смещений, поэтому там каждой программе для DOS подменяют адрес перехода по прерыванию, после чего он становится 32-разрядным. Вот почему отладчик AfdPro может видеть в первых 1024 байтах памяти одни адреса, а инструкциями

```
mov  ax, es:[bx]   ;смещение
mov  dx, es:[bx+2] ;сегмент
```

в регистры ax и dx будут записаны совсем другие. Но большинство программ этого не заметят, продолжая жить так, как будто ими управляет система DOS.

Прерывания, с которыми мы только что познакомились, называются *программными*. Встретив инструкцию int 21h, процессор прерывает как бы сам себя. Но бывают так называемые *аппаратные* прерывания, чей источник лежит вне процессора. Сигналы этих прерываний поступают процессору от внешних устройств, таких как клавиатура или жесткий диск. Многие вещи эти устройства способны выполнить самостоятельно, без участия процессора. Но иногда процессор им все-таки нужен. Например, при нажатии клавиши нужно прочитать введенный символ и запомнить его в буфере. Но процессор один, а устройств, которым он нужен, много. Поэтому устройство, когда это ему необходимо, должно заставить процессор работать на себя, послав ему запрос на прерывание и его номер. Если прерывания разрешены, процессор запоминает в стеке адрес возврата и регистр флагов, получает адрес программы, обрабатывающей прерывание, делает что требуется, пока не встретит инструкцию iret, возвращающую его к прерванной работе.

Теперь становится понятно, почему прерывание требует сохранять в стеке не только адрес возврата, но и регистр флагов. Ведь *аппаратное* прерывание, в отли-

чие от программного, возникает в случайный момент времени. И может, например, попасть между операцией сравнения и инструкцией перехода[1]:

```
cmp ax, 0
<прерывание>
jnz label
```

Результат сравнения находится в регистре флагов, и если его не сохранить, процессор после обработки аппаратного прерывания пойдет не тем путем и в результате безнадежно запутается.

[1] Перед выполнением прерывания процессор обязательно завершит текущую инструкцию.

ГЛАВА 10 Полезности



Управление потоком

Нужно думать не о том, что нам может пригодиться, а только о том, без чего мы не сможем обойтись.

Джером К. Джером. «Трое в лодке, не считая собаки»

В этой главе пойдет речь именно о том, без чего большинство программистов может обойтись. Но не обходится. Это различные «улучшения» инструкций процессора, предлагаемые ассемблером.

Чтобы стало ясно, о чем речь, вспомним программу из листинга 4.2 (см. раздел «Переходы» главы 4), где нужно было направить процессор по разным путям, в зависимости от величины некой переменной. Фрагмент ассемблерной программы, где у процессора есть два варианта действий, был таким, как в листинге 10.1.

Листинг 10.1. Пример ветвлений в ассемблере

```
cmp digit,0
jnz nzero
invoke  WriteConsoleA, stdout, ADDR z, \
        zsize, ADDR cWritten, NULL
jmp exit
nzero:
invoke  WriteConsoleA, stdout, ADDR nz, \
        nzsize, ADDR cWritten, NULL
exit:   invoke  ExitProcess, 0
```

Ключевую роль здесь играет инструкция jnz nzero, отправляющая процессор к метке nzero, когда переменная digit не равна нулю, и позволяющая процессору выполнить следующую инструкцию, если digit равна нулю.

Вместе с безусловным переходом jmp инструкция jnz организует две ветви вычислений. В одном случае программа выведет на экран равно нулю, в другом — не равно нулю.

Это ветвление выглядит не очень красиво и не очень понятно, где одна ветвь, где другая. Поэтому в ассемблере введены специальные директивы .IF, .ELSE .ENDIF, с помощью которых программа из листинга 4.2 может быть переписана так, как показано в листинге 10.2.

Листинг 10.2. Организация ветвлений с помощью условных директив

```
.386
.model  flat, stdcall
option  casemap:none
include    \myasm\include\windows.inc
include    \myasm\include\kernel32.inc
includelib \myasm\lib\kernel32.lib
.data
z  db  "равно нулю", 13, 10
zsize dd  ($-z)
nz db  "не равно нулю", 13, 10
nzsize dd  ($-nz)
digit dd  1
stdout   dd  ?
cWritten dd  ?
.code
start:
invoke  GetStdHandle, STD_OUTPUT_HANDLE
mov stdout, eax
.IF digit == 0
invoke  WriteConsoleA, stdout, ADDR z, \
        zsize, ADDR cWritten, NULL
.ELSE
invoke  WriteConsoleA, stdout, ADDR nz, \
        nzsize, ADDR cWritten, NULL
.ENDIF
invoke  ExitProcess, 0
end start
```

Здесь проверку, равно ли нулю число digit, выполняет директива .IF digit == 0. Если digit равно нулю, выполняется первая ветвь программы, чьи инструкции

расположены между директивой .IF и директивой .ELSE. Если же digit не равно нулю, выполняется вторая ветвь между .ELSE и .ENDIF.

Нужно отчетливо понимать, что не существует таких инструкций процессора, как .IF и .ELSE. Встретив эти директивы, ассемблер превратит их в настоящие инструкции процессора, поэтому программа в окне отладчика будет выглядеть совсем не так, как в листинге 10.2. Рисунок 10.1, где изображен фрагмент программы, соответствующий конструкции .IF .ELSE .ENDIF, показывает, что ассемблер превратил эти директивы в обычные команды процессора cmp, jnz, jmp, такие же, как в листинге 10.1.

```
0040100C  . 833D 23304000 CMP DWORD PTR DS:[403023],0
00401013  .v75 1F         JNZ SHORT BRANCH2.00401034
00401015  . 6A 00         PUSH 0
00401017  . 68 2B304000   PUSH BRANCH2.0040302B
0040101C  . FF35 0C304000 PUSH DWORD PTR DS:[40300C]
00401022  . 68 00304000   PUSH BRANCH2.00403000
00401027  . FF35 27304000 PUSH DWORD PTR DS:[403027]
0040102D  . E8 32000000   CALL <JMP.&kernel32.WriteConsoleA>
00401032  .vEB 1D         JMP SHORT BRANCH2.00401051
00401034  > 6A 00         PUSH 0
00401036  . 68 2B304000   PUSH BRANCH2.0040302B
0040103B  . FF35 1F304000 PUSH DWORD PTR DS:[40301F]
00401041  . 68 10304000   PUSH BRANCH2.00403010
00401046  . FF35 27304000 PUSH DWORD PTR DS:[403027]
0040104C  . E8 13000000   CALL <JMP.&kernel32.WriteConsoleA>
00401051  > 6A 00         PUSH 0
00401053  . E8 00000000   CALL <JMP.&kernel32.ExitProcess>
```

Рис. 10.1. Так видит отладчик программу из листинга 10.2

Директивы .IF .ELSE .ENDIF, с которыми мы только что познакомились, по-разному оцениваются программистами. Многие осуждают их за то, что они превращают ассемблер в подобие языка высокого уровня, такого как Си, где нет однозначного соответствия между текстом программы и выданной компилятором последовательностью инструкций процессора. А это соответствие считается одним из преимуществ ассемблера перед другими языками. Ассемблер потому и прост, что совершенно не абстрактен, он «поет о том, что видит», то есть позволяет по тексту программы однозначно ска-

зать, какую последовательность команд исполнит процессор.

На мой взгляд, в этих упреках есть своя правда, хотя и до директив .IF .ELSE .ENDIF мы уже вступили на скользкую дорожку, ведущую к языкам высокого уровня, когда согласились использовать директиву invoke для запуска процедуры и терпели своеволие ассемблера, добавлявшего в процедуру пролог push ebp, mov ebp. esp и эпилог leave (см. раздел «Своеволие ассемблера» главы 3). В защиту директив можно сказать, что они не нарушают однозначного соответствия между исходным текстом на ассемблере и соответствующей последовательностью инструкций процессора. Они просто отдаляют одно от другого. И решать, использовать ли директивы, организующие ветвление в программе, каждый должен сам. Впрочем, эти директивы нужно по крайней мере знать, потому что они часто встречаются во многих исходных текстах.

Поэтому продолжим знакомство с ними, вернее, с различными условиями в директиве .IF. Одно мы уже знаем. Знак == означает «равно». Другие условия интуитивно понятны, а тем, кто знает язык Си, еще и привычны:

```
!=  не равно
>   больше
>=  больше или равно
<   меньше
<=  меньше или равно
```

Глядя на эти условия, стоит вспомнить, что в ассемблере есть два типа сравнений — для чисел со знаком и без. Так вот, директивы .IF .ELSE .ENDIF по умолчанию считают числа беззнаковыми, то есть ассемблер поставит вместо .IF eax <0 инструкцию jb, а для условия .IF eax > 0 поставит инструкцию ja. Чтобы заставить ассемблер использовать инструкции сравнения чисел со

знаком jg и jl[1], нужно пометить одно из сравниваемых чисел оператором SDWORD PTR (для двойного слова), SWORD PTR (для слова) или же SBYTE PTR (для байта). Так, например, директива .IF SDWORD PTR digit > 0 превратится в инструкцию jle (если меньше или равно — перейти), а директива .IF digit > 0 станет инструкцией jbe, которая работает с числами без знака.

Круженье

Кроме директив, помогающих программе ветвиться, есть еще директивы, организующие циклы. Мы уже встречались с циклами, заданными инструкцией loop. Теперь попробуем заменить loop в листинге 4.3 (см. раздел «Повторение» главы 4) директивами .WHILE .ENDW, с помощью которых вывод на экран десяти чисел подряд будет выглядеть так, как в листинге 10.3

Листинг 10.3. Организация цикла с помощью директив .while .endw

```
...
mov ecx. 10
.WHILE ecx !=0
 push    ecx
 push    edx
 invoke  wsprintf. ADDR buf. ADDR ifmt. edx
 invoke  WriteConsoleA. stdout. ADDR buf. \
 BSIZE. ADDR cWritten. NULL
 invoke  WriteConsoleA. stdout. ADDR crlf.\
         2. ADDR cWritten. NULL
 pop     edx
 inc     edx
 pop     ecx
 dec     ecx
.ENDW
...
```

[1] С ними мы уже познакомились в разделе «Ввод» главы 5.

Перед циклом .WHILE в регистр ecx посылается число 10. А дальше проверяется, равен ли ecx нулю. Если да — цикл завершается, если нет — совершает новый оборот. Естественно, ecx нужно менять внутри цикла, чтобы тот не крутился вечно. Поэтому перед .ENDW стоит инструкция dec ecx.

Кроме директив .WHILE .ENDW для организации цикла можно использовать похожие директивы .REPEAT .UNTIL, отличающиеся тем, что проверка, от результата которой зависит продолжение цикла, делается не в начале, а в конце. Цикл, показанный в листинге 10.3, организуется директивами .REPEAT .UNTIL так:

```
mov ecx. 10
.REPEAT
...
dec     ecx
.UNTIL ecx == 0
```

Обратите внимание: такой цикл выполняется хотя бы раз, потому что условие выхода проверяется в самом его конце. По сравнению с циклом, организованном директивой .WHILE, здесь все наоборот: цикл прекращается, когда условие в директиве .UNTIL истинно (в нашем примере — когда ecx обратится в ноль).

Задача 10.1. Посмотрите с помощью отладчика OllyDbg, как ассемблер реализует циклы .WHILE .ENDW и .REPEAT .UNTIL.

Макросы

В программах часто повторяются одни и те же фрагменты, такие, например, как завершение работы в системе DOS:

```
mov  ah. 4ch   ;завершить программу
int  21h
```

Смысл этих строк довольно туманен, да и выписывать их каждый раз не хочется. И было бы здорово заставить ассемблер при встрече какого-нибудь короткого, ясного слова, например Quit (выход), вставлять в текст программы две строки, приведенные выше.

Чтобы решить эту задачу, в ассемблере есть *макросы*, позволяющие назвать одним словом сколь угодно длинный текст. Программу из листинга 8.1, выводящую на экран фразу Не могу молчать!, можно переписать с использованием макросов так, как показано в листинге 10.4.

Листинг 10.4. Пример использования макросов

```
Quit macro
     mov   ah, 4ch
     int   21h
endm
LDisp macro line
      mov   dx, offset line
      mov   ah, 09
      int   21h
endm
.8086
.MODEL  small
option casemap:none
.stack 100
.data
hello  db "Не могу молчать!", 0dh, 0ah, '$'
.code
start:
mov  dx,@stack
mov  ss,dx
mov  dx,@data
mov  ds,dx     ;регистр данных
LDisp hello    ;вывод на экран
Quit           ;уходим
end  start
```

Макрос Quit определяется в самом начале программы так:


```
Quit macro
     mov   ah, 4ch
     int   21h
endm
```

Сначала идет имя макроса, затем слово macro, составляющие его заголовок, затем тело макроса, состоящее из двух строк, и признак конца макроса endm. После того как макрос определен, ассемблер заменит каждое слово Quit, встреченное в программе, двумя строками

```
mov   ah, 4ch
int   21h
```

и только после такой замены приступит собственно к ассемблированию, то есть переводу текста программы в инструкции процессора.

Как видим, замена строк

```
mov   ah, 4ch
int   21h
```

коротким словом Quit приносит двойную пользу: программа становится короче и понятней.

Но часто такая замена невозможна, из-за того, что тело макроса содержит параметр, который может меняться в разных местах программы. Например, строки

```
mov   dx, offset hello
mov   ah, 09
int   21h
```

выводят на экран сообщение, помеченное как hello, но в программе может быть много сообщений и писать для каждого собственный макрос просто глупо. Вместо этого пишется макрос с формальным параметром line (см. листинг 10.4):

```
LDisp macro line
      mov   dx, offset line
      mov   ah, 09
      int   21h
endm
```

При вызове макроса вместо формального параметра ставится фактический. В программе из листинга 10.4 строка

```
LDisp hello
```

обрабатывается следующим образом: формальный параметр line всюду в теле макроса заменяется фактическим hello, и затем преображенное тело макроса вставляется в текст программы вместо строки LDisp hello. Так что ассемблер видит перед собой три строки

```
mov   dx, offset hello
mov   ah, 09
int   21h
```

и уже *их* преобразует в инструкции процессора.

В рассмотренном примере у макроса был один параметр. Но их может быть сколько угодно. При вызове таких макросов параметры разделяются запятыми. В качестве примера создадим макрос, читающий файл в системе DOS. Эту задачу выполняет функция 3fh прерывания 21h. Для нормальной работы ей необходимы три параметра: в регистре bx должен быть *хендл* файла — по сути его номер в операционной системе, который программа узнает при создании файла. Этот хендл похож на дескриптор файла, возвращаемый процедурой CreateFile Windows API. Второй параметр — число читаемых байтов — должен быть в регистре cx и, наконец, третий параметр — смещение буфера, куда читаются байты из файла. Оно хранится в регистре dx (смещение должно быть указано относительно сегмента ds). С учетом сказанного, макрос, читающий файл, может выглядеть так:

```
Read macro FHandle, NOfBytes, Buff
mov     bx, FHandle
mov     cx, NOfBytes
mov     dx, .o Buff
mov     ah, 3fh
int     21h
endm
```


Если вызвать этот макрос строкой

```
Read Handle, 16d, PackBuff,
```

то формальные параметры заменятся фактическими, и 16 байт из файла, чей хендл хранится в переменной Handle, будут прочитаны в буфер PackBuff.

Иногда при вызове макроса не хочется указывать все параметры. В нашем примере может случиться так, что хендл уже хранится в bx прямо перед вызовом макроса. На этот случай существует директива ifnf (*If Not Blank* — если не пуст[1]). С ее помощью макрос можно переписать следующим образом:

```
Read    macro    FHandle, NOfBytes, Buff
ifnb     <FHandle>
mov      bx, FHandle
endif
mov      cx, NOfBytes
mov      dx, .o Buff
mov      ah, 3fh
int      21h
endm
```

При этом смысл его будет таким: если формальный параметр Fhandle указан, он будет заменен фактическим параметром, который отправится в регистр bx. То есть строки

```
ifnb     <FHandle>
mov      bx, FHandle
endif
```

превратятся в

```
mov      bx, FHandle
```

Если же макрос вызывается без параметра Fhandle, то посылать в регистр bx нечего (подразумевается, что хендл уже там) и строки ifnb ...endif будут просто пропущены.

[1] Существует, конечно, и противоположная директива ifb (If Blank — если пуст): ifb <параметр> ... endif

Иными словами, ассемблер, встретив вызов макроса

```
Read        .16d. PackBuff
```

поймет, что первого параметра нет, и потому не станет посылать его в регистр bx.

Как видите, макросы очень похожи на процедуры. У них, как и у процедур, есть параметры, а вызов макроса напоминает запуск процедур директивой invoke. Но это сходство обманчиво. Ведь процедуры по-настоящему отделены от основной программы, они хранят параметры и свои локальные переменные в стеке. Макросы же только прикидываются процедурами, а на самом деле они принадлежат основной программе и могут быть источником ошибок. Кроме того, макросы вставляются в программу при каждом вызове, а потому занимают больше памяти. Но у макросов есть и преимущества: ими легче манипулировать, материал, из которого сделан макрос, более податлив. Кроме того, вызов процедуры требует процессорного времени, чтобы сохранить в стеке передаваемые параметры. Макрос получает свои параметры сразу. Поэтому там, где требуется высокая скорость вычислений, лучше использовать макрос.

ГЛАВА 11 Ассемблер и другие языки



В этой короткой главе пойдет речь о месте ассемблера в программировании. До сих пор мы писали программы целиком на ассемблере, потому что именно ему посвящена эта книга. Но в реальной жизни так поступают только самые «упертые» фанатики, не желающие знать (а зачастую и не знающие) других языков.

Поступая так, они ощущают свое превосходство над простыми пользователями Паскаля или Бейсика. И совершенно напрасно. Ведь ассемблер, если честно,— первобытный, первоначальный язык, верный девизу: «что вижу — о том пою». В ассемблере каждая инструкция понятна и подробно описана. И если существуют на свете сложные языки, то это скорее C++. Так что ассемблер не стоит изучать только потому, что это «круто». Ассемблер нужен совсем для другого.

Прежде всего, без знания ассемблера невозможно понять, как работает операционная система, как она делит ресурсы между программами и как хранит данные в своих служебных областях.

Ассемблер необходим при создании программ, взаимодействующих с аппаратурой. Это могут быть драйверы устройств, работающих с Windows или DOS.

Далее, ассемблер нужен программисту, чтобы понять, почему программа работает неправильно. Современные компиляторы очень хороши, но и они ошибаются. И если программист не понимает, в чем дело, он велит компилятору дать «отчет о проделанной работе» — показать листинг программы на ассемблере.

Наконец, ассемблер необходим там, где от программы требуется большая скорость. Вычислительная мощь

современных компьютеров чудовищно велика и стремительно растет. Но сложность решаемых задач растет еще быстрее. Вот почему производительности даже самых мощных компьютеров не хватает. Когда обнаруживается, что программа на языке высокого уровня работает правильно, но слишком долго, программист прежде всего пытается найти узкие места в программе, для чего она подвергается профилированию: специальная программа следит; сколько времени потрачено на определенные участки программы, сколько раз вызываются те или иные процедуры.

Как правило, профилирование выявляет узкие места программы, на которые тратится большая часть времени процессора. Вот эти места и следует переписать на ассемблере, потому что квалифицированный программист делает это лучше, чем компилятор языка высокого уровня.

В этой книге мы почти не интересовались временем выполнения инструкций процессора и не пытались писать быстро работающие программы, потому что это сложная, обширная тема, требующая отдельной книги. Но понять, как вообще сочетается ассемблер и языки высокого уровня, мы сможем.

Представим себе, что написана программа на языке Си[1], в которой функция xchg меняет местами две целочисленные переменные a и b (листинг 11.1).

Листинг 11.1. Простая программа на языке Си

```
#include <stdio.h>
void xchg(int *a.int *b):
int main(){
int a=2. b=3:
xchg(&a.&b):
```

продолжение ↗

[1] Тем, кто не знает Си, могу рекомендовать книгу А. Крупника «Изучаем Си», Питер, 2001.

Листинг 11.1. *(продолжение)*

```
printf("a= %d b= %d\n",a,b);
return 0;
}
void xchg(int *a,int *b){
  int tmp;
  tmp=*a;
  *a=*b;
  *b=tmp;
}
```

Как и положено в языке Си, функция xchg получает два указателя на int.

А теперь поставим перед собой задачу научиться сочетать функции, написанные на Си, и функции, написанные на ассемблере. Проще всего это сделать, подсмотрев, как компилятор транслирует функцию на язык ассемблера. Разумеется, каждый компилятор делает это по-своему, поэтому попробуем поработать с тем, что оказалось под рукой — компилятором Borland C++ версии 5.5.1[1].

В компиляторе Borland C++ выдачей листинга на ассемблере управляет ключ -S. Чтобы получить этот листинг, сохраним функцию в отдельном файле xchg.c:

```
--------------файл xchg.c------------------void xchg(int
*a,int *b){
  int tmp;
  tmp=*a;
  *a=*b;
  *b=tmp;
}
```

и запустим из оболочки FAR компилятор:

```
bcc32 -c -S xchg.c
```

ключ -c в командной строке означает, что на выходе создается только объектный файл xchg.obj, компоновщик не запускается. А ключ -S командует компилятору создать ассемблерный листинг функции.

[1] Этот компилятор бесплатен и его можно найти на ftp-сайте фирмы Borland ftp://ftpd.borland.com/download/bcppbuilder/freecommandLinetools.exe.

После запуска компилятора в папке, где хранится исходный текст функции hchg.c, появятся объектный файл xchg.obj и ассемблерный листинг xchg.asm. Открыв его, увидим кучу непонятных директив, меток, начинающихся знаком вопроса, и комментариев. Это текст на ассемблере, созданный компилятором и потому не очень подходящий человеку. Но если его не пугаться, в нем можно выделить строки, непосредственно относящиеся к нашей функции (листинг 11.2).

Листинг 11.2. Функция xchg, переведенная на язык ассемблера

```
_xchg  proc  near
?live1@0:
   ;
   ;  void xchg(int *a,int *b){
   ;
push      ebp
mov       ebp, esp
push      ebx
mov       edx, dword ptr [ebp+12]
mov       eax, dword ptr [ebp+8]
   ;
   ; int tmp;
   ; tmp = *a;
   ;
?live1@16: ; EAX = a, EDX = b
@1:
mov       ecx, dword ptr [eax]
   ;
   ;*a = *b;
   ;
?live1@32: ; EAX = a, EDX = b, ECX = tmp
mov       ebx,dword ptr [edx]
mov       dword ptr [eax],ebx
   ;
   ; *b = tmp;
   ;
?live1@48: ; EDX = b, ECX = tmp
```

продолжение ➩

Листинг 11.2 *(продолжение)*

```
mov        dword ptr [edx].ecx
    :
    : }
    :
?livel@64: :
@2:
    pop        ebx
    pop        ebp
    ret
_xchg  endp
```

В этом отрывке сочетаются инструкции ассемблера и комментарии, в которых показаны соответствующие инструкции языка Си. Начинается функция хорошо известным нам прологом

```
push       ebp
mov        ebp. esp
```

После него отсчет параметров, переданных в стек, идет относительно ebp. Параметры эти занимают привычные нам места [ebp+8] и [ebp+12] и переписываются в регистры edx, eax:

```
mov        edx. dword ptr [ebp+12]
mov        eax. dword ptr [ebp+8]
```

Комментарий, приведенный чуть ниже, показывает, что в регистр eax попадает параметр a, в регистр же edx записывается параметр b. Это значит, что первым в стек загружается параметр b, затем a.

Следующий комментарий говорит нам, что временной переменной служит регистр ecx. Дальнейшие инструкции совершенно понятны:

```
mov        ecx. dword ptr [eax]:tmp = *a
mov        ebx. dword ptr [edx]:*a = *b
mov        dword ptr [eax]. ebx:
mov        dword ptr [edx]. ecx:*b = *tmp
...
ret
```


Они, кстати, раскрывают тайну указателей в языке Си, показывая, что это простые адреса.

Завершается функция выталкиванием из стека двух регистров ebx, ebp и, конечно, возвратом ret. Регистр ebx выталкивается потому, что в начале функции он был сохранен в стеке. Почему же не был сохранен ecx? Очевидно, таковы правила компилятора: регистром ecx он не дорожит, а ebx использует для каких-то своих целей и потому не допускает его порчи внутри функции. Список регистров, которые нужно сохранять в стеке, можно найти в документации к компилятору. Но можно просто получить ассемблерный листинг сложной функции, использующей все регистры, и посмотреть, какие из них сохраняются в стеке.

Зная «законы компилятора», легко «выпотрошить и выбросить вон» созданную им функцию, а взамен написать свою, которая, возможно, будет работать быстрее. В случае с компилятором Borland C++ последовательность действий будет такой:

1. Создается программа на языке Си, где объявлена функция, которую нужно переписать на ассемблере. В нашем случае она выглядит так:

```
-----------------main.c-----------------
#include <stdio.h>
void xchg(int *a.int *b):
int main(){
int a=2.b=3:
xchg(&a.&b):
printf("a= %d b= %d\n".a.b):
return 0:
}
----------------------------------------
```

2. Создается функция на языке ассемблера xchg.asm. Как принять параметры внутри функции и какие регистры сохранить, подскажет компилятор, если

создать «муляж» функции на языке Си и получить ассемблерный листинг.

Оба файла передаются компилятору: `bcc32 main.c hchg.asm`, который создаст файл main.exe[1].

Ну а дальше начинается самое главное: нужно так подобрать инструкции процессора, чтобы они выполнялись быстрее созданных компилятором языка высокого уровня. Для каждого процессора фирмы Intel эта задача решается по-своему, потому что время выполнения одной и той же инструкции у разных процессоров различно. Чтобы справиться с этой задачей, нужно хорошо знать устройство процессоров и того, что их окружает. Ведь скорость выполнения программ часто определяется не самим процессором, а его взаимодействием с компьютерной памятью. Но все это — тема других, гораздо более толстых книг.

[1] Перед запуском компилятора ВСС придется установить на компьютере 32-битовый ассемблер фирмы Borland tasm32.exe. Эта программа не распространяется бесплатно, но ее легко можно найти в Интернете или в одной из файлообменных сетей (Gnutella, Kazaa или Edonkey).

Крупник Александр Борисович

Изучаем Ассемблер

Главный редактор	*Е. Строганова*
Заведующий редакцией	*А. Кривцов*
Руководитель проекта	*Ю. Суркис*
Литературный редактор	*А. Чижова*
Художник	*М. Соколинская*
Корректор	*В. Листова*
Верстка	*М. Богер*

Лицензия ИД № 05784 от 07.09.01.
Подписано к печати 25.10.04. Формат 84×108 32. Усл. п. л. 13,44.
Доп. тираж 3000. Заказ [illegible]
ООО «Питер Принт», 194044, Санкт-Петербург, пр. Б. Сампсониевский, д. 29а.
Налоговая льгота — общероссийский классификатор продукции
ОК 005-93, том 2; 95 3005 — литература учебная.

Отпечатано с готовых диапозитивов
в ООО «Северо-Западный печатный двор»
г. Гатчина. ул. Солодухина, 2